ESP32-P4

ESP-VISION Programming Guide



Release 2026.11.14-2-gace7ca10bd
Espressif Systems
Nov 14, 2026

Table of contents

Table of contents i

1 Introduction 3
- 1.1 What Is ESP-VISION . . . 3
- 1.2 Quick Experience . . . 3
- 1.3 Key Features . . . 3
- 1.4 Supported Boards . . . 3

2 Chip and Board Support 5
- 2.1 Supported Development Boards . . . 5
- 2.2 Chip Versus Board Capabilities . . . 7
- 2.3 Standard MicroPython Features . . . 7
- 2.4 WebREPL and EV-MUX Network Control . . . 7

3 Get Started 9
- 3.1 Prerequisites . . . 9
- 3.2 Build, Flash, and Monitor . . . 9
- 3.3 Common idf.py Commands . . . 10
- 3.4 Application-only Flashing . . . 10
- 3.5 Run Your First Script . . . 10

4 Concepts 11
- 4.1 MicroPython Language Fundamentals . . . 11
  - 4.1.1 Execution Model . . . 11
  - 4.1.2 Basic Syntax . . . 12
  - 4.1.3 Main Data Structures . . . 13
  - 4.1.4 Design Principles for Microcontrollers . . . 14
  - 4.1.5 MicroPython and CPython . . . 14
- 4.2 Packages and Deployment . . . 15
  - 4.2.1 Module and Package Layout . . . 16
  - 4.2.2 Package Management in the Current Firmware . . . 16
  - 4.2.3 Packages on the Filesystem . . . 16
  - 4.2.4 Packages Embedded in Firmware . . . 17
  - 4.2.5 Choosing a Deployment Method . . . 17
  - 4.2.6 Import Priority and Version Control . . . 18
- 4.3 The Image Model . . . 18
  - 4.3.1 Pixel formats . . . 18
  - 4.3.2 Color spaces . . . 19
  - 4.3.3 The frame buffer and `fb_alloc` . . . 19
  - 4.3.4 Region of interest (ROI) . . . 19
- 4.4 Image Processing . . . 19
  - 4.4.1 A typical processing pipeline . . . 19
  - 4.4.2 Thresholding and segmentation . . . 20
  - 4.4.3 Neighborhood (convolution) filters . . . 20
  - 4.4.4 Rank and edge-preserving filters . . . 21
  - 4.4.5 Morphology . . . 21
  - 4.4.6 Histograms and statistics . . . 21

4.4.7 Feature detection . . . 21
4.5 AI Inference . . . 22
4.5.1 Model Runtimes . . . 22
4.5.2 Model Files and Quantization . . . 22
4.5.3 Inference flow . . . 22
4.5.4 Pre-processing . . . 23
4.5.5 Post-processing . . . 23
4.5.6 Memory and performance . . . 24
4.6 The Camera Pipeline . . . 24
4.6.1 From sensor to image . . . 25
4.6.2 Per-board backends . . . 25
4.6.3 Frame sizes . . . 26
4.7 Codecs and Streaming . . . 26
4.7.1 Still-image codecs: JPEG and PNG . . . 26
4.7.2 Recording sequences: ImageIO . . . 26
4.7.3 Video codec: H.264 . . . 26
4.7.4 Network streaming: RTSP . . . 27
4.7.5 Host preview: USB CDC . . . 27
4.8 Startup Sequence . . . 27
4.8.1 Hard Reset and Soft Reset . . . 28
4.8.2 ESP-VISION Startup Order . . . 28
4.8.3 First Boot and Flash Filesystem . . . 30
4.8.4 Using boot.py . . . 30
4.8.5 Using main.py . . . 30
4.8.6 REPL and Recovery . . . 31

5 Storage, Filesystems and Data Protection **33**
5.1 Media and filesystem ownership . . . 33
5.2 Optional volumes and automatic mounting . . . 34
5.3 USB with two volumes . . . 35
5.4 Failure policy: preserve data and keep control available . . . 35
5.5 One writer, safe publication and power loss . . . 36
5.6 Backup and restore policy . . . 37
5.7 Explicit recovery of an unmounted Flash volume . . . 37
5.8 Firmware installation and acceptance . . . 38

6 API Reference **39**
6.1 MicroPython APIs . . . 39
6.1.1 Documentation Baseline . . . 39
6.1.2 Language and Built-in APIs . . . 39
6.1.3 Standard and Micro Libraries . . . 40
6.1.4 JSON and Time Example . . . 40
6.1.5 Hardware and ESP32 APIs . . . 40
6.1.6 GPIO and PWM Example . . . 40
6.1.7 Networking APIs . . . 41
6.1.8 Connect to Wi-Fi . . . 41
6.1.9 Filesystem and Runtime APIs . . . 41
6.1.10 List Storage and Check Memory . . . 41
6.1.11 Structuring a Complex Application . . . 42
6.1.12 Threads and Native Tasks . . . 44
6.1.13 Check Availability on a Firmware . . . 44
6.2 sensor –Camera . . . 44
6.2.1 Camera Initialization and Continuous Capture . . . 44
6.2.2 Image Orientation and Camera Status . . . 45
6.2.3 Temporarily Stop Capture . . . 45
6.2.4 IPA Runtime Control . . . 45
6.2.5 Reading IPA Statistics . . . 46
6.2.6 Manual ISP Control . . . 47

6.2.7 Constants . . . 49
6.2.8 Functions . . . 49
6.2.9 Classes . . . 51
6.3 image –Image Processing . . . 51
6.3.1 Drawing and Color-Blob Tracking . . . 51
6.3.2 Filtering and Binary Segmentation . . . 51
6.3.3 QR Code Recognition . . . 52
6.3.4 Line, Segment, and Circle Detection . . . 52
6.3.5 Encode and Save an Image . . . 52
6.3.6 Constants . . . 53
6.3.7 Functions . . . 55
6.3.8 Classes . . . 56
6.4 display –LCD Output . . . 68
6.4.1 Camera Preview . . . 68
6.4.2 Drawing into a Reusable Image . . . 68
6.4.3 Position, Scale, and Crop . . . 69
6.4.4 Backlight and Cleanup . . . 69
6.4.5 Classes . . . 69
6.5 lcd_touch –Touch Input . . . 70
6.5.1 Read Touch Coordinates . . . 70
6.5.2 Ownership and Cleanup . . . 71
6.5.3 Classes . . . 71
6.6 espdl –Model Inference . . . 71
6.6.1 Button-Confirmed Single-Object Tracking . . . 71
6.6.2 Raw Output Tensors . . . 73
6.6.3 Object Detection . . . 74
6.6.4 PicoDet . . . 74
6.6.5 PedestrianDetect . . . 74
6.6.6 Image Classification . . . 74
6.6.7 Pose Estimation . . . 75
6.6.8 Adjust Thresholds at Runtime . . . 75
6.6.9 Result tuples . . . 75
6.6.10 Functions . . . 75
6.6.11 Classes . . . 76
6.7 tflite –Model Inference . . . 79
6.7.1 Basic Usage . . . 79
6.7.2 Callable Inputs . . . 79
6.7.3 Post-processing . . . 79
6.7.4 Classes . . . 80
6.8 image.ImageIO –Image Stream . . . 80
6.8.1 Record to Storage . . . 80
6.8.2 Replay a Recorded Stream . . . 80
6.8.3 Use an In-Memory Stream . . . 81
6.8.4 Constants . . . 81
6.8.5 Classes . . . 81
6.9 h264 –H.264 Encoding . . . 82
6.9.1 Encode One Frame . . . 82
6.9.2 Record a Raw H.264 Stream . . . 83
6.9.3 Resource and Throughput Considerations . . . 83
6.9.4 Classes . . . 83
6.10 rtsp –RTSP Streaming . . . 84
6.10.1 Capture, Encode, and Stream . . . 84
6.10.2 Client and Queue Behavior . . . 85
6.10.3 Shutdown . . . 85
6.10.4 Classes . . . 85
6.11 uvc —USB webcam output . . . 86
6.12 storage –Diagnostics and Confirmed Recovery . . . 87
6.12.1 Functions . . . 87

7 How-To Guides **89**
7.1 Build vision applications with AI . . . 89
7.1.1 Firmware-matched knowledge . . . 89
7.1.2 Connect your AI assistant . . . 89
7.1.3 Connect a board and run an application . . . 90
7.1.4 Everyday tasks . . . 91
7.1.5 Improve the result through conversation . . . 93
7.1.6 Tools behind the conversation . . . 93
7.2 Add a New Python Module . . . 93
7.2.1 Overview . . . 94
7.2.2 1. Create the Binding Source . . . 94
7.2.3 2. Register qstrs If Needed . . . 94
7.2.4 3. Wire the Source into the Build . . . 94
7.2.5 4. Add a Type Stub . . . 95
7.2.6 5. Document the Module . . . 95
7.2.7 6. Build and Verify . . . 95
7.3 Contribution and Review . . . 95
7.3.1 Sync and merge process . . . 95
7.4 Customize Firmware Features . . . 96
7.4.1 Choose the Customization Scope . . . 96
7.4.2 Customize ESP-VISION Python Modules . . . 96
7.4.3 Customize Image Algorithms . . . 96
7.4.4 Customize Standard MicroPython Features . . . 96
7.4.5 Configure Memory Placement . . . 97
7.4.6 Customize Frozen Python Code . . . 97
7.4.7 Customize Board Services and Optional Components . . . 97
7.4.8 Build and Verify . . . 97
7.5 Publish qualified firmware . . . 98
7.5.1 Package and URL contract . . . 98
7.5.2 Immutable manifest and current pointer . . . 98
7.5.3 Build-derived Wi-Fi capability . . . 99
7.5.4 Optional single-bin layout . . . 99
7.5.5 MCP and release qualification . . . 99
7.5.6 Release procedure . . . 99
7.5.7 Iris firmware files . . . 100
7.6 Introduce a New Model . . . 103
7.6.1 1. Obtain or Convert the Model . . . 103
7.6.2 2. Copy the Model to Board Storage . . . 103
7.6.3 3. Pick the Right API . . . 103
7.6.4 4. Run ESP-DL Inference . . . 104
7.6.5 5. Decode ESP-DL Outputs in Python . . . 104
7.6.6 6. Run TFLite Micro Inference . . . 104
7.6.7 7. Optional: Profiling and Validation . . . 105
7.7 Train an ESPDet Pico Model . . . 105
7.7.1 1. Prepare the ESP-Detection Environment . . . 105
7.7.2 2. Prepare the Dataset . . . 106
7.7.3 3. Train ESPDet Pico . . . 106
7.7.4 4. Prepare Calibration Data . . . 107
7.7.5 5. Quantize and Export .espdl . . . 107
7.7.6 6. Add Model Metadata . . . 108
7.7.7 7. Copy the Model to Board Storage . . . 108
7.7.8 8. Run on ESP-VISION . . . 109
7.7.9 9. Validate and Tune . . . 109
7.8 Add a New Board . . . 110
7.8.1 MicroPython Port Side . . . 110
7.8.2 ESP-VISION Side . . . 110
7.8.3 Build and Flash . . . 111
7.9 EV-MUX v4 Host Integration Entry . . . 111

7.10 Agent Code Generation Guide . . . 111
7.10.1 Dependency Rules . . . 111
7.10.2 All-in-One Output . . . 111
7.10.3 Device Communication and Acceptance . . . 112

8 Solution Architecture **113**
8.1 Layered Overview . . . 113
8.2 Capture-to-Output Data Flow . . . 115
8.3 Communication Subsystem . . . 115
8.4 Source Tree . . . 116
8.5 Board Composition . . . 116
8.6 Chip-Dependent Sources . . . 116
8.7 MicroPython Overlay . . . 116

9 Communication System and EV-MUX **119**
9.1 The complete path at a glance . . . 120
9.2 Recommended reading order . . . 120
9.2.1 Architecture, Execution Contexts and Standard I/O . . . 120
9.2.2 Physical interfaces, connection generations and build configuration . . . 128
9.2.3 USB Interfaces, Configuration and Disk Names . . . 133
9.2.4 Session ownership, acquisition and recovery . . . 135
9.2.5 Frame format, logical channels and RPC protocol . . . 140
9.2.6 Interactive Python, scripts and file operations . . . 146
9.2.7 Preview, debug capture, backpressure and statistics . . . 153
9.2.8 Host integration, Wi-Fi configuration and recovery . . . 159
9.2.9 LAN discovery protocol and board defaults . . . 165
9.2.10 Device Core integration guide . . . 168
9.2.11 Communication acceptance, closed-loop tests and troubleshooting . . . 168
9.3 Six distinct concepts . . . 175
9.4 Current implementation boundaries . . . 175

10 Device Core host SDK **177**
10.1 Application API reference . . . 177
10.1.1 Creation and byte transport . . . 178
10.1.2 device.discover(options) . . . 179
10.1.3 device.connect(options) . . . 179
10.1.4 device.disconnect() . . . 179
10.1.5 device.inspect(request) . . . 179
10.1.6 device.configure(request) . . . 180
10.1.7 device.control(request) . . . 180
10.1.8 device.python.input(data) . . . 181
10.1.9 device.python.execute(request) . . . 181
10.1.10 Device file operations . . . 181
10.1.11 device.capture(request) . . . 182
10.1.12 device.on(event, handler) . . . 183
10.1.13 Concurrency, cancellation and errors . . . 183
10.1.14 Firmware checks and updates . . . 184
10.1.15 Selecting a firmware version . . . 186
10.1.16 Immutable release snapshots and single-BIN updates (0.4.0) . . . 186
10.2 Advanced export API reference . . . 187
10.2.1 @esp-vision/device-core . . . 187
10.2.2 @esp-vision/device-core/evmux . . . 188
10.2.3 @esp-vision/device-core/evrepl . . . 188
10.2.4 @esp-vision/device-core/session . . . 190
10.2.5 @esp-vision/device-core/actions . . . 193
10.2.6 @esp-vision/device-core/bindings . . . 205
10.2.7 @esp-vision/device-core/firmware . . . 206
10.2.8 @esp-vision/device-core/node . . . 208
10.2.9 @esp-vision/device-core/discovery . . . 209

10.2.10 @esp-vision/device-core/discovery/node . . . 209
10.2.11 Integration boundaries . . . 209
10.2.12 FirmwareFlashSession.flash result . . . 210
10.2.13 @esp-vision/device-core/browser . . . 210
10.2.14 Release readers in @esp-vision/device-core/firmware (0.5.1) . . . 210
10.2.15 Browser flash startup . . . 210
10.3 Integrating the Device Core npm Package . . . 211
10.3.1 Source, versions and installation . . . 211
10.3.2 Application, Core and transport responsibilities . . . 211
10.3.3 Standalone Node TCP example . . . 213
10.3.4 Browser Web Serial example . . . 214
10.3.5 Files and interactive Python . . . 214
10.3.6 Wi-Fi configuration and status . . . 215
10.3.7 Discovery and identity checks . . . 215
10.3.8 Preview, logs and capture . . . 216
10.3.9 Reset, flashing and publication . . . 217
10.3.10 Firmware update application entry . . . 217
10.3.11 Browser ROM flashing . . . 217
10.4 Device management CLI . . . 218
10.4.1 Installation and host requirements . . . 218
10.4.2 Discovery and inspection . . . 218
10.4.3 Configuration, control, and firmware . . . 219
10.4.4 Python execution . . . 219
10.4.5 Device files and capture . . . 220
10.4.6 Batch and watch . . . 220
10.4.7 Output, timeouts, and cancellation . . . 221
10.5 Installation and compatibility . . . 221
10.6 Architecture and starting point . . . 221
10.7 Licensing . . . 223

11 Project Relationships **225**
11.1 ESP-VISION, MicroPython, and OpenMV . . . 225
11.2 Dependency Layers . . . 225

12 Licensing **227**
12.1 License Inventory . . . 227
12.2 How to Interpret the Licenses . . . 229
12.3 OpenMV and GPL Code Paths . . . 229
12.4 Adding Third-Party Code . . . 229

Python Module Index **231**

Python Module Index **231**

Index **233**

Index **233**

ESP-VISION is a Low-Code Edge AI & Computer Vision Framework for Espressif SoCs. It deeply integrates essential capabilities including camera capture, image processing, video encoding and decoding, network transmission, model deployment, and AI inference, while providing unified and standardized Python APIs that enable developers to rapidly build edge applications combining visual capture, intelligent recognition, display, and media streaming.

This guide is organized as follows.

Chapter 1

Introduction

1.1 What Is ESP-VISION

ESP-VISION is a Low-Code Edge AI & Computer Vision Framework for Espressif SoCs. It deeply integrates essential capabilities including camera capture, image processing, video encoding and decoding, network transmission, model deployment, and AI inference, while providing unified and standardized Python APIs that enable developers to rapidly build edge applications combining visual capture, intelligent recognition, display, and media streaming.

1.2 Quick Experience

Visit the ESP-VISION website for quick access to Web IDE, MCP setup resources, examples, and related project resources.

1.3 Key Features

- Unified camera, image, display, video encoding, preview, and streaming APIs across supported chips and boards.
- Image processing capabilities covering drawing, filtering, color tracking, feature detection, QR codes, barcodes, and AprilTags.
- ESP-DL-powered object detection, pose estimation, and image classification, plus TensorFlow Lite Micro support for `.tflite` model execution.
- Efficient C/C++ foundation components work closely with on-chip multimedia peripherals and hardware acceleration modules to deliver high-performance, real-time application execution.
- Development through a VSCode-based host tool or Web IDE, with firmware builds managed through `idf.py`.

1.4 Supported Boards

ESP-VISION supports boards based on ESP32-P4, ESP32-S3, and ESP32-S31. See *Chip and Board Support* for the full board list and the chip-specific modules and constraints.

See *Get Started* to build and flash the firmware, and *Solution Architecture* for how the pieces fit together.

Chapter 2

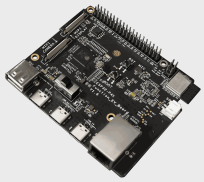
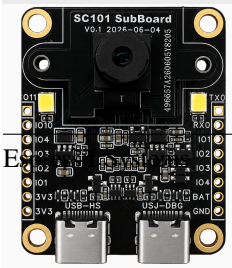
Chip and Board Support

ESP-VISION documentation is built separately for each supported chip. Select the chip used by the board firmware so that the documented modules and APIs match the default build.

2.1 Supported Development Boards

Note: Current standard prebuilt ESP32-P4 firmware supports silicon revisions v3.1–v3.99. This is the chip revision reported by esptool, not a board or module silkscreen version. Check the exact image's revision bounds before flashing; a board name or a generic v3.x label alone is insufficient.

No prebuilt firmware is provided here for ESP32-P4 v1.3. Users who need it must build and validate their own firmware: select the matching board, enable `CONFIG_ESP32P4_SELECTS_REV_LESS_V3` in menuconfig, choose a minimum revision compatible with the chip, and rebuild both the bootloader and application. IDF's pre-v3 and v3.x targets are mutually incompatible. Do not force-flash a v3.x image or patch its revision header to bypass checks.

Picture	Board	Chip	ESP-VISION support
ESP-VISION	ESP32-S31-CHATBOT	ESP32-S3	**Supported** | Modules: sensor, image, display, lcd_touch, espdl, tflite, imageio, storage | MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT | Vision: pixel/math, filters, geometry, template matching, QR code, AprilTag
	ESP32-S31-Korvo-1	ESP32-S3	**Supported** | Modules: sensor, image, display, lcd_touch, espdl, tflite, imageio, uvc, storage | MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT | Vision: pixel/math, filters, geometry, template matching, QR code, AprilTag
	ESP32-S31-MOSAICO	ESP32-S3	**Supported** | Modules: sensor, image, display, lcd_touch, espdl, tflite, imageio, uvc, storage | MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT | Vision: pixel/math, filters, geometry, template matching, QR code, AprilTag
	ESP32-P4X-EYE	ESP32-P4	**Supported** | Modules: sensor, image, display, lcd_touch, espdl, tflite, imageio, h264, rtsp, uvc, storage, barcode | MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT | Vision: pixel/math, filters, geometry, template matching, QR code, AprilTag
	ESP32-P4X-Function-EV-Board	ESP32-P4	**Supported** | Modules: sensor, image, display, lcd_touch, espdl, tflite, imageio, h264, rtsp, uvc, storage, barcode | MicroPython: network, WLAN, I2C, SPI, UART, PWM, WDT | Vision: pixel/math, filters, geometry, template matching, QR code, AprilTag
	ESP32-P4X-	ESP32-P4	**Supported** | Modules: sensor, image, dis-

Note: The support summary is generated from `mpconfigport.h`, each board `mpconfigboard.h` and `imlib_config.h`, and the chip-specific module selection.

2.2 Chip Versus Board Capabilities

The selected chip controls chip-level source selection. For example, `h264` and `rtsp` are added by `micropython.cmake` only when `IDF_TARGET` is `esp32p4`. A board profile can further enable or disable features such as the ZXing-C++ barcode backend through `boards/<BOARD>/board.cmake`.

The API navigation and chip-specific symbol conditions are generated from these build files. This makes `micropython.cmake`, `boards/*/port/mpconfigboard.cmake`, and board `board.cmake` files authoritative for ESP-VISION API availability.

2.3 Standard MicroPython Features

Standard MicroPython features use a separate configuration path: `overlay/micropython/ports/esp32/mpconfigport.h` plus each board's `mpconfigboard.h`. The table above indexes the explicitly enabled baseline features from those files, while additional conditions can depend on chip capabilities and firmware configuration.

The current board headers disable Bluetooth and ESP-NOW; the S31 board additionally disables `machine.ADC` and `machine.ADCBlock`. A board must still provide suitable networking hardware and firmware configuration before `network.WLAN` can connect.

Because these conditions are not determined solely by the chip, the chip selector filters the ESP-VISION API reference but does not represent a complete standard MicroPython module manifest. Build-environment compatibility details are maintained in the project README.

The chip documentation describes the defaults shared by the current board profiles. A custom board or modified build option can therefore differ from the published chip page. When that happens, the firmware configuration is authoritative.

2.4 WebREPL and EV-MUX Network Control

Traditional MicroPython WebREPL is not supported by the current ESP-VISION EV-MUX runtime, including ESP-IDF v5.5.6 builds. Some builds enable WebSocket and the internal `_webrepl` component. The upstream `webrepl` Python service is absent from some board profiles, including ESP32_P4X_EYE; ESP32_S3_EYE includes it in its manifest. Neither its presence nor the compile flags establish terminal or file-transfer compatibility with EV-MUX.

EV-MUX routes Python input, output and files through one authorized owner, with link, lease and VM-epoch checks. A traditional WebREPL `dupterm` service does not implement this ownership contract. Adding its Python package or mirroring raw terminal bytes is not a supported workaround. Use *Application API reference* with a TCP transport for network control. The retained `example/07-Network/00-WebREPL/webrepl.py` entry exits with this explanation before changing Wi-Fi or opening a listener.

Chapter 3

Get Started

3.1 Prerequisites

- ESP-IDF **v5.5.6** for ESP32-P4/S3 boards, or **release/v6.1** for ESP32-S31 boards. Source the matching checkout's `export.sh` so that `idf.py` is on `PATH`.
- A board listed in *Chip and Board Support* for the selected chip.

Warning: CI and released firmware qualify only these board/IDF combinations. v5.5.6 is a pinned patch release; S31 follows the maintained `release/v6.1` branch. Other IDF versions may change ESP-IDF or managed-component APIs, fail to build, or change runtime behavior and the available MicroPython APIs. Adapting source, overlays or dependencies and repeating hardware verification may be necessary. The project README describes the optional MicroPython feature differences.

Note: When building on Windows, install Visual Studio Build Tools before running `idf.py`. In the Visual Studio Build Tools installer, select the "Desktop development with C++" workload and make sure the "MSVC v143 - VS 2022 C++ x64/x86 build tools" component is selected.

3.2 Build, Flash, and Monitor

Clone the repository with submodules, then use the board-aware `idf.py` extension from the repository root:

```
git clone --recursive https://github.com/espressif/esp-vision.git esp-vision
cd esp-vision
```

```
idf.py --board ESP32_P4X_EYE -p /dev/ttyACM0 build flash monitor
```

The command first runs `prepare-micropython`: it verifies that `lib/micropython` is checked out at the pinned MicroPython v1.28.0 commit, exports a clean MicroPython build copy under `build/micropython/`, applies `overlay/micropython/` to that copy, then projects each `boards/<BOARD>/port/` onto the copy's `ports/esp32/boards/<BOARD>/`. `lib/micropython` is never dirtied.

3.3 Common idf.py Commands

Command	Description
`idf.py --board <BOARD> build`	Build firmware for a board.
`idf.py --board <BOARD> app`	Build the application for a board.
`idf.py --board <BOARD> -p <PORT> app-flash`	Build as needed and flash only the application at its configured partition offset.
`idf.py --board <BOARD> -p <PORT> flash`	Build and flash firmware.
`idf.py --board <BOARD> -p <PORT> monitor`	Open the serial monitor.
`idf.py --board <BOARD> menuconfig`	Open menuconfig.
`idf.py --board <BOARD> -p <PORT> erase-flash`	Erase flash.
`idf.py --board <BOARD> clean`	Clean board build output.
`idf.py --board <BOARD> fullclean`	Remove the complete build directory for the selected board.

3.4 Application-only Flashing

`app` and `app-flash` are supported for every supported board/ESP-IDF combination, including ordinary firmware with Iris disabled. `app-flash` uses the current build's partition table to select the application address; it does not write the bootloader, partition table, OTA metadata or data partitions. The board must already have the matching layout and boot selection. It does not provision a blank device or migrate partitions.

Use the selected board's ROM download port and an explicit build directory for that firmware configuration. For example, with a matching ordinary P4X EYE installation:

```
idf.py --board ESP32_P4X_EYE -B "$ESP_VISION_WORK_DIR/build/ESP32_P4X_EYE" \
  -p "$PORT" app-flash
```

Ordinary `flash` still writes the full build's flashing manifest. Only when `ESP_VISION_ENABLE_IRISFW=ON` does root `flash` automatically select `app-flash`; the retained Mosaico layout uses `0x210000`, which must not be assumed for other configurations. See *Direct application flashing* for Iris-specific setup and verification.

3.5 Run Your First Script

After flashing, connect over the REPL and try the camera. Each *API Reference* module page links the runnable `example/` scripts for that API.

Chapter 4

Concepts

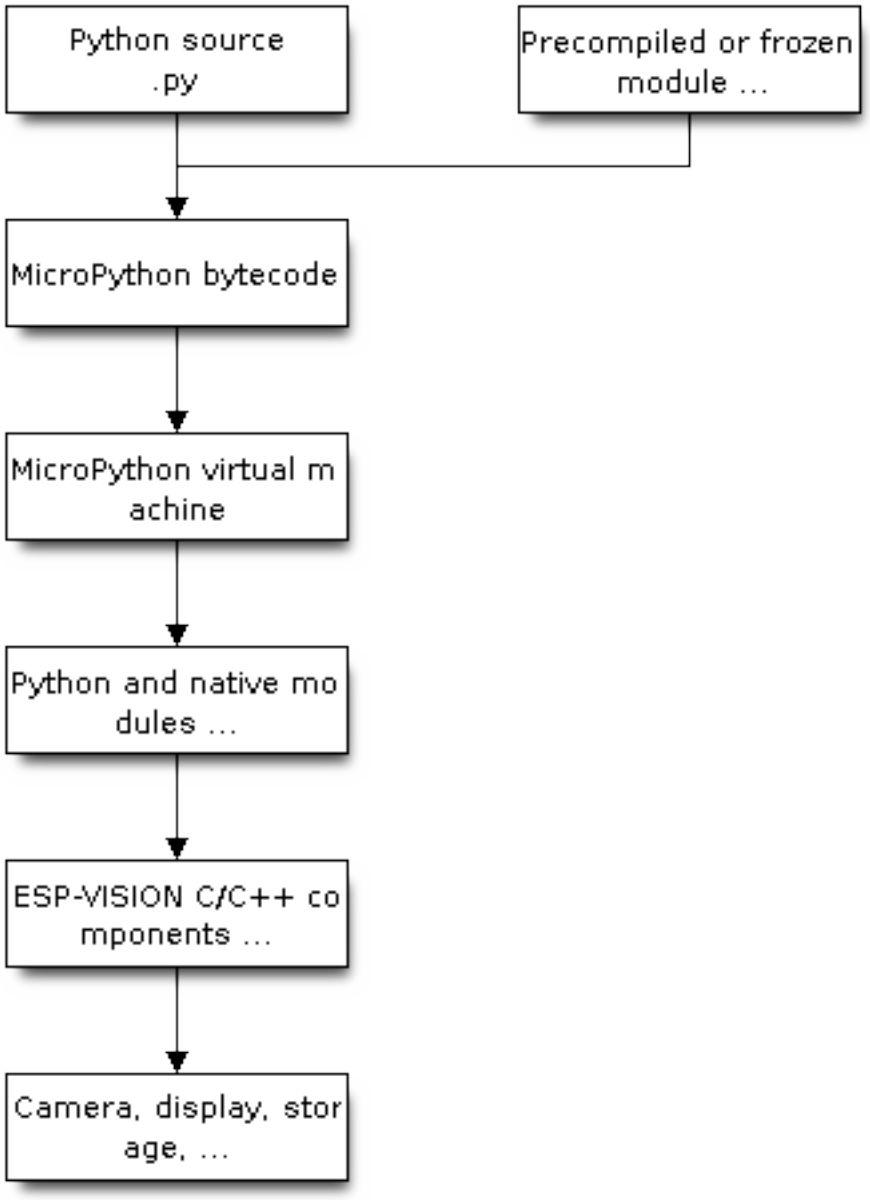
This section explains the ideas behind the ESP-VISION API: how images are represented in memory, how the image-processing algorithms work, how neural network inference runs on the device, and how frames move through the camera and codec pipelines. Read it alongside the *API Reference* to understand *why* each function behaves the way it does.

4.1 MicroPython Language Fundamentals

MicroPython is a compact implementation of the Python language designed to run directly on microcontrollers. ESP-VISION uses MicroPython v1.28.0 as its application language while retaining C/C++ and ESP-IDF components for hardware access and performance-critical vision work. This chapter introduces the language model needed to read and write ESP-VISION applications; the authoritative details remain in the MicroPython v1.28.0 language and implementation reference.

4.1.1 Execution Model

Upstream MicroPython aims to implement Python 3.4 syntax with selected features from later Python versions. ESP-VISION builds the ESP32 port at the `EXTRA_FEATURES` configuration level, with the on-device compiler, garbage collector, MPZ long integers, single-precision floating point, scheduler, VFS, and persistent bytecode loading enabled. Source code remains dynamically typed Python, but it is compiled to compact bytecode and executed by the MicroPython virtual machine. Native modules connect Python objects to ESP-VISION C/C++ components, ESP-IDF drivers, and chip hardware:



Code loaded from `.py` is compiled on the device and consumes RAM during compilation. Precompiled `.mpy` files and modules frozen into the firmware reduce runtime compilation work; immutable constants in frozen modules can remain in Flash. Python still owns application policy and object lifetimes, while native components implement operations such as frame capture, image processing, codecs, and AI inference.

4.1.2 Basic Syntax

The current ESP-VISION configuration uses indentation to define blocks and supports the familiar Python forms for assignment, expressions, conditions, loops, functions, classes, exceptions, context managers, comprehensions, generators, imports, and `async`/`await`. Names are dynamically bound to objects, so a declaration does not specify a C-style storage type:

```
from micropython import const

_MIN_PIXELS = const(80)

class Detection:
    def __init__(self, label, score):
        self.label = label
        self.score = score

    def accepted(self, threshold=0.5):
        return self.score >= threshold

def select_results(results):
    selected = []
    for label, score in results:
        item = Detection(label, score)
        if item.accepted():
            selected.append(item)
    return selected

try:
    detections = select_results((("person", 0.91), ("chair", 0.42)))
    for item in detections:
        print(item.label, item.score)
except (ValueError, TypeError) as error:
    print("invalid result:", error)
finally:
    print("processing complete")
```

`const()` is a MicroPython extension that lets the compiler substitute a constant value and can reduce bytecode and global-dictionary overhead. Type annotations may be useful to developers and editors, but MicroPython does not use them to provide C-style static typing or automatic memory layout.

4.1.3 Main Data Structures

The choice between mutable and immutable objects matters on a microcontroller because creating, resizing, or concatenating objects allocates heap memory:

Type	Mutability	ESP-VISION usage
`None` and `bool`	Immutable	Represent absence and state flags without defining custom sentinel objects.
`int` and `float`	Immutable	Store counters, coordinates, thresholds, and scores. Repeated arithmetic can create new objects; avoid floating-point work in hard interrupt context.
`str`	Immutable	Store text, paths, labels, and JSON keys. Concatenation creates a new string, so avoid building large strings every frame.
`bytes`	Immutable	Store compact binary constants, encoded packets, and read-only model or protocol data.
`bytearray`	Mutable	Provide reusable binary buffers for peripheral I/O and packet assembly without allocating a new object for every operation.
`list`	Mutable	Store ordered results whose size changes. `append()` and growth may allocate memory, so bound or reuse lists in sustained frame loops.
`tuple`	Immutable	Represent fixed records such as rectangles and detection results. Constant tuples can be more memory-efficient than repeatedly created lists.
`dict`	Mutable	Represent configuration and shared scalar state. Adding keys can resize the table; create the expected structure during initialization.
`set`	Mutable	Perform membership and uniqueness checks when the additional hash-table memory is justified.
`range`	Immutable	Describe integer iteration without constructing a list of all values.
`memoryview` and `array`	View / mutable	Access typed or sliced buffer data with fewer copies; the referenced buffer must remain alive and must not be resized while a view is active.

Object references are shared rather than copied automatically. Assigning `b = a` makes both names refer to the same mutable list, dictionary, image, or buffer; use an explicit copy only when independent data is required. This is especially important for *`image.Image`* objects backed by reusable camera frame buffers.

4.1.4 Design Principles for Microcontrollers

MicroPython prioritizes portability, interactive development, compact firmware integration, and direct hardware access over complete CPython library coverage. Its garbage-collected heap makes dynamic application code practical, but RAM, Flash, stack space, and allocation latency remain finite. The MicroPython guide for constrained devices recommends reducing repeated object creation, preallocating buffers, moving constants and reusable modules into frozen bytecode, and monitoring the heap with `gc.mem_free()` and `gc.mem_alloc()`.

For ESP-VISION applications, initialize modules, models, buffers, and stable dictionaries before entering the frame loop. Reuse `bytearray` and image buffers, keep per-frame temporary data bounded, and avoid retaining a camera-backed image after the next `sensor.snapshot()` unless it has been copied. Use Python to express orchestration and product behavior; leave deterministic real-time work, ISR handling, bulk pixel processing, codecs, and inference kernels in native components.

4.1.5 MicroPython and CPython

In this documentation, "Python" in a runtime comparison means CPython 3.x, the reference implementation commonly used on PCs and servers. The exact MicroPython feature set is firmware-dependent, so this table describes MicroPython v1.28.0 as configured by ESP-VISION: all maintained board manifests freeze `asyncio`, the ESP32 port enables `_thread` with a GIL, and hardware APIs remain subject to chip and board configuration.

Area	ESP-VISION MicroPython	CPython
Primary environment	Runs inside MCU firmware with direct access to peripherals and board services.	Runs as an operating-system process on PCs, servers, and larger embedded systems.
Language syntax	Implements Python 3.4 syntax with selected later features; most common control flow, functions, classes, exceptions, generators, and asynchronous syntax are available.	Tracks the current Python language specification and generally introduces new syntax first.
Type system	Dynamically typed; annotations do not make execution statically typed.	Dynamically typed; annotations are consumed mainly by tools and optional libraries.
Standard library	Provides a resource-oriented subset selected at firmware build time, plus `machine`, `micropython`, `esp`, and `esp32` hardware/runtime modules.	Provides the full CPython standard library for the installed version, without a standard MCU peripheral API.
Package deployment	Modules are copied to storage, precompiled as `.mpy`, or frozen into firmware; desktop `pip` and arbitrary binary wheels must not be assumed.	Commonly uses `pip`, virtual environments, source distributions, and platform wheels.
Memory model	Uses a small garbage-collected heap shared with firmware resources; allocation failures and fragmentation must be considered explicitly.	Uses a much larger process heap and reference counting with cyclic garbage collection.
Concurrency	Provides frozen `asyncio`, enabled `_thread` with a GIL, and scheduled GPIO IRQ and Timer callbacks; peripheral availability still depends on the selected chip and board.	Provides `asyncio`, threads, processes, and extensive operating-system synchronization facilities.
Performance path	Calls native C/C++ modules and hardware accelerators for demanding operations; Python is best used for orchestration.	Can use optimized extension modules, JIT-enabled alternative runtimes, or multiprocessing, but usually has greater CPU and memory resources.
Compatibility details	Some built-in behavior, exception text, introspection, module contents, and edge cases differ to reduce code size and RAM usage.	Defines the reference behavior against which MicroPython differences are documented.
Portability	Pure Python code using the supported subset is often portable; hardware modules and resource assumptions are device-specific.	Pure Python code is portable across supported operating systems when its dependencies are available.

Do not decide portability from syntax alone. Check imported modules, memory use, filesystem paths, concurrency assumptions, and hardware access. The upstream MicroPython and CPython differences list documents known behavioral differences, while *MicroPython APIs* describes the inherited APIs and concurrency model available in ESP-VISION.

4.2 Packages and Deployment

MicroPython organizes reusable code as modules and packages, but an embedded product must also decide where that code is stored and how it is updated. This chapter describes package management and deployment for the MicroPython v1.28.0 version pinned by ESP-VISION; see the upstream package management and manifest references for the complete generic behavior.

4.2.1 Module and Package Layout

A module is normally a `.py` source file or a precompiled `.mpy` file. A package is a directory containing an `__init__.py` file and one or more modules or subpackages, so a reusable vision pipeline can use the following layout:

```
my_vision/
    __init__.py
    pipeline.py
    transport.py
```

Applications import package members with standard Python syntax:

```
from my_vision.pipeline import VisionPipeline

pipeline = VisionPipeline()
```

4.2.2 Package Management in the Current Firmware

MicroPython uses `mip` rather than CPython's `pip` to install packages from micropython-lib, package indexes, URLs, or a local `package.json` description. The default ESP-VISION manifests do not currently embed the device-side `mip` module, so `import mip` is not available in the standard firmware; use `mpremote` on the development host to install packages into the device filesystem:

Important: The default ESP-VISION firmware does not support device-side `mip`. Running `import mip` or `mip.install()` on the device raises `ImportError`. The host-side `mpremote mip` command remains available and is the default package installation method.

```
mpremote connect <PORT> mip install --target=/lib <PACKAGE>
mpremote connect <PORT> mip install --target=/lib <PACKAGE>@<VERSION>
mpremote connect <PORT> mip install --target=/lib ./package.json
```

A `package.json` file describes the files and dependencies that `mip` installs at runtime or during provisioning. It is not a firmware manifest and does not determine which modules are compiled into ESP-VISION. Package compatibility must be checked against MicroPython v1.28.0 and the selected chip architecture because MicroPython packages can depend on firmware features or architecture-specific `.mpy` files.

Device-side installation with `mip.install()` requires a customized firmware that explicitly includes `mip` together with the required networking and TLS support. Host-side installation with `mpremote` is the recommended default because it does not increase the product firmware image or require the device to download code from the network.

4.2.3 Packages on the Filesystem

ESP-VISION mounts the internal Flash filesystem at `/` and adds `/lib` to `sys.path`. Install reusable packages under `/lib`; modules in the current directory, frozen modules, and packages under `/lib` can then be imported without changing the application code.

An SD card is mounted at `/sdcard` when enabled, but its package directory is not added to `sys.path` automatically. Add it explicitly before importing packages stored there:

```
import sys

sys.path.append("/sdcard/lib")
from my_vision.pipeline import VisionPipeline
```

Source `.py` files are compiled when imported, which consumes RAM for bytecode and adds import latency. Pre-compiled `.mpy` files avoid compilation on the device and reduce source exposure, but their bytecode still occupies RAM after loading and they must match the MicroPython version and architecture.

4.2.4 Packages Embedded in Firmware

Firmware manifests select Python modules and packages that are compiled and frozen into the firmware image. ESP-VISION keeps board manifests in `boards/<BOARD>/manifest.py` and currently uses them to freeze the port modules, ESP-VISION initialization modules, `asyncio`, and selected board-specific features.

For example, place `my_vision/__init__.py` and its modules under `boards/<BOARD>/packages`, then add the package to that board's manifest:

```
package(
    "my_vision",
    base_path="$(ESP_VISION_ROOT)/boards/<BOARD>/packages",
)
```

The lower-level `freeze()` function can also freeze a directory or selected modules, while `include()` and `require()` reuse manifest fragments and packages from configured manifest libraries. Reconfigure and rebuild the selected board after changing its manifest:

```
idf.py --board <BOARD> reconfigure
idf.py --board <BOARD> build
idf.py --board <BOARD> flash
```

4.2.5 Choosing a Deployment Method

Use the expected update frequency and product ownership of a package to select its deployment method:

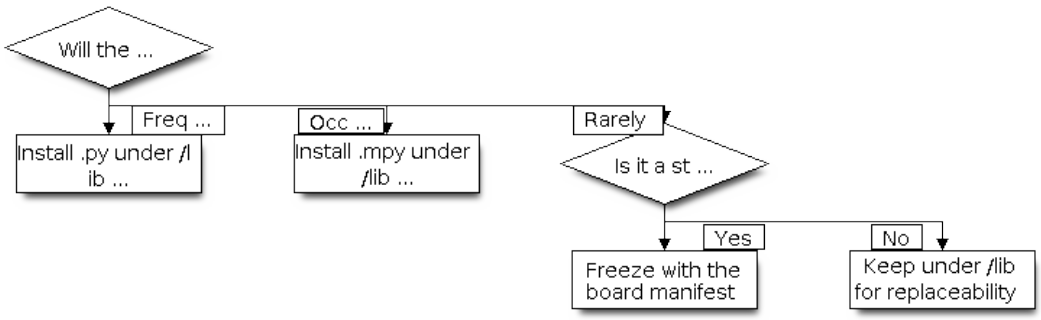


Fig. 1: Package deployment decision

Table 1: Deployment method comparison

Property	`.py` under `/lib`	`.mpy` under `/lib`	Frozen in firmware
Storage	Writable filesystem	Writable filesystem	Firmware Flash image
Import behavior	Compiled when imported	Loaded without source compilation	Executes directly from frozen bytecode
Runtime RAM	Bytecode and objects use RAM	Bytecode and objects use RAM	Frozen bytecode and constants can remain in Flash
Update method	Replace files or use `mpremote mip`	Replace compatible `.mpy` files	Rebuild and flash firmware
Primary use	Development and frequently updated application code	Replaceable modules with lower import overhead	Stable product dependencies and startup modules

4.2.6 Import Priority and Version Control

The current firmware searches the current directory, frozen modules through `.frozen`, and then `/lib`. A filesystem package with the same import name therefore does not override a frozen package during normal import; remove the frozen package from the manifest and rebuild the firmware, or use a different package name.

Keep board manifests and frozen package revisions in source control so firmware builds are reproducible. Use `package.json` to describe files and dependencies installed by `mip`, and use `manifest.py` to define modules embedded during the firmware build; these formats serve different deployment stages and are not interchangeable.

For ESP-VISION products, freeze stable framework extensions and startup dependencies, while keeping product scripts, configuration, and field-updatable modules under `/lib`. Large AI models, images, and video assets should normally remain data files in Flash, SD card, or external storage instead of being embedded as Python package data, so they can be updated and mapped by the appropriate runtime component.

4.3 The Image Model

Every vision operation in ESP-VISION reads or writes an *`image.Image`*: a rectangular grid of pixels plus a small header describing its width, height, and pixel format. Understanding how those pixels are laid out, which color spaces the algorithms use, and how the underlying memory is managed is the key to writing fast, correct scripts.

4.3.1 Pixel formats

The pixel format determines how many bytes each pixel occupies and how its value is interpreted.

Format	Bytes/pixel	Description
`BINARY`	1 bit	Black/white. Used for masks and the output of *`image.Image.binary()`*.
`GRAYSCALE`	1	8-bit intensity (0-255). The format most filters and detectors prefer.
`RGB565`	2	16-bit color: 5 bits red, 6 bits green, 5 bits blue.
`BAYER`	1	Raw single-channel mosaic straight from the sensor; debayered on demand.
`YUV422`	2	Luminance/chrominance packing, common in camera/codec pipelines.
`JPEG` / `PNG`	varies	Compressed byte streams. JPEG is produced by *`image.Image.to_jpeg()`*; PNG codec support is not enabled in the current firmware.

RGB565 packs a color into 16 bits, trading color precision for half the memory of RGB888. Green gets the extra bit because the human eye is most sensitive to it. *`image.Image.get_pixel()`* can return either the packed value or an `(r, g, b)` tuple expanded back to 8 bits per channel.

4.3.2 Color spaces

Although pixels are *stored* as grayscale or RGB565, color thresholding works in the **LAB** color space. LAB separates lightness (L) from two color-opponent axes (A: green-red, B: blue-yellow), so a threshold expressed in LAB is far more robust to brightness changes than one in RGB. This is why a color threshold is a six-tuple `(l_min, l_max, a_min, a_max, b_min, b_max)` while a grayscale threshold is just `(min, max)`.

The module-level conversion helpers (*`image.rgb_to_lab()`*, *`image.lab_to_rgb()`*, *`image.rgb_to_grayscale()`*, and the rest of the `*_to_*` family) expose these conversions for a single pixel, which is handy when computing thresholds offline.

4.3.3 The frame buffer and `fb_alloc`

Embedded memory is scarce, so ESP-VISION avoids per-frame heap allocation. *`sensor.snapshot()`* returns an image backed by a reusable **frame buffer**: the next `snapshot()` overwrites it. If you need to keep a frame around (for example, to compare against a later frame in *`image.Image.difference()`*), make an explicit `img.copy()`.

Many algorithms need scratch memory that lives only for the duration of a call. These use a stack-like allocator, `fb_alloc`, that carves temporary buffers out of the frame-buffer region and frees them all at once when the operation returns. This is why heavy methods do not fragment the heap, and why you should prefer reusing one image over allocating new ones in a hot loop.

4.3.4 Region of interest (ROI)

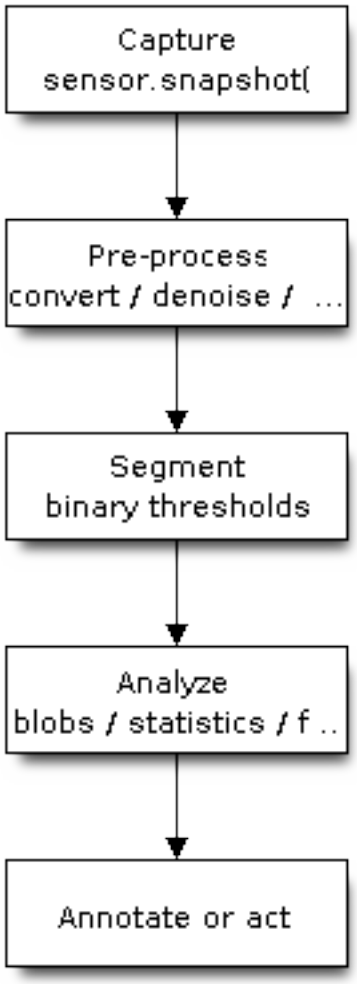
Most analysis and conversion methods accept an `roi=(x, y, w, h)` keyword that restricts the operation to a sub-rectangle of the image. Working on an ROI is both faster (fewer pixels) and more selective (ignore irrelevant areas). The ROI is always expressed in the source image's pixel coordinates.

4.4 Image Processing

The vision algorithms in the *image –Image Processing* module come from OpenMV's `imlib` library, maintained in this repository as the `components/imlib` IDF component. They fall into a few families: point and geometric transforms, neighborhood filters, statistical analysis, and feature detection. This page explains what each family does and when to reach for it.

4.4.1 A typical processing pipeline

Most color-tracking or inspection scripts follow this processing flow:



Pre-processing can combine `image.Image.to_grayscale()`, `image.Image.gaussian()`, and `image.Image.histeq()`; segmentation commonly uses `image.Image.binary()`; analysis then uses `image.Image.find_blobs()`, `image.Image.get_statistics()`, or a feature detector before the application draws results or takes an action.

4.4.2 Thresholding and segmentation

`image.Image.binary()` converts the image into a mask by testing each pixel against a list of color thresholds (see *The Image Model* for why thresholds are expressed in LAB). A pixel is "on" if it falls inside *any* of the supplied thresholds, which lets you track several colors at once. The companion `image.Image.find_blobs()` runs an 8-connected component labeling pass over the same thresholds and returns one `image.blob` per connected region, with centroid, bounding box, pixel count, orientation, and shape descriptors such as roundness and solidity.

4.4.3 Neighborhood (convolution) filters

These replace each pixel with a function of its `(2*ksize+1)` square neighborhood:

- `image.Image.mean()` –box average; fast blur.

- *image.Image.gaussian()* –weighted average using a Pascal-triangle (binomial) approximation of a Gaussian; the standard noise-reduction blur. Set `unsharp=True` to sharpen instead.
- *image.Image.laplacian()* –second-derivative edge response; set `sharpen=True` to add it back to the original for edge enhancement.
- *image.Image.morph()* –apply an arbitrary integer kernel. `mul` and `add` scale and bias the result; the default `mul` is `1/sum(kernel)` so the image brightness is preserved.

All of these share `threshold`/`offset`/`invert` keywords that turn the filter into an adaptive thresholding operation in a single pass.

4.4.4 Rank and edge-preserving filters

Rank filters sort the neighborhood instead of averaging it, which removes noise without blurring edges as much:

- *image.Image.median()* –the percentile (default 0.5) value; excellent against salt-and-pepper noise.
- *image.Image.mode()* –the most common value.
- *image.Image.midpoint()* –a `bias` blend between the min and max.
- *image.Image.bilateral()* –a Gaussian blur weighted by both spatial distance (`space_sigma`) and color similarity (`color_sigma`), so it smooths flat regions while keeping edges crisp.

4.4.5 Morphology

image.Image.erode() and *image.Image.dilate()* grow or shrink the set pixels of a (usually binary) image using a square structuring element; *image.Image.open()* and *image.Image.close()* combine them to remove specks or fill small holes. The `threshold` argument controls how many neighbors must be set, generalizing the classic binary operators.

4.4.6 Histograms and statistics

image.Image.get_histogram() bins pixel values per channel; the returned *image.histogram* can compute Otsu thresholds (*image.histogram.get_threshold()*), percentiles, and full statistics. *image.Image.get_statistics()* returns mean, median, mode, standard deviation, quartiles, and min/max in one call, which is useful for auto-tuning thresholds at runtime.

4.4.7 Feature detection

Higher-level detectors find geometric structure:

- *image.Image.find_lines()* and *image.Image.find_circles()* use the Hough transform; their `threshold` is the minimum accumulator score and the `*_margin` keywords merge near-duplicate results.
- *image.Image.find_line_segments()* uses Edge Drawing Lines; its `threshold` is the minimum Sobel gradient magnitude, while `merge_distance` and `max_theta_diff` control segment merging.
- *image.Image.find_rects()* locates quadrilaterals (useful for fiducials and screens).
- *image.Image.find_qrcodes()* and *image.Image.find_apriltags()* detect and decode the corresponding marker types; AprilTags can additionally return 6-DoF pose when camera intrinsics are supplied.

The current ESP32-P4 board profiles additionally provide *image.Image.find_barcodes()* through the ZXing-C++ backend.

Note: GPL-licensed OpenMV code paths are disabled in this build (`OMV_NO_GPL=1`), so a small number of upstream algorithms are intentionally unavailable. The per-board `imlib_config.h` selects which optional algorithms are compiled in; a method that is not enabled raises an exception at call time.

4.5 AI Inference

ESP-VISION provides on-device neural-network execution through *espdl – Model Inference*, which uses ESP-DL, and *tflite – Model Inference*, which uses TensorFlow Lite Micro for TensorFlow Lite `.tflite` models. The available APIs range from task-oriented helpers to general model execution, and model-specific preprocessing or post-processing can live in module bindings, helper code, or Python scripts depending on the model family.

4.5.1 Model Runtimes

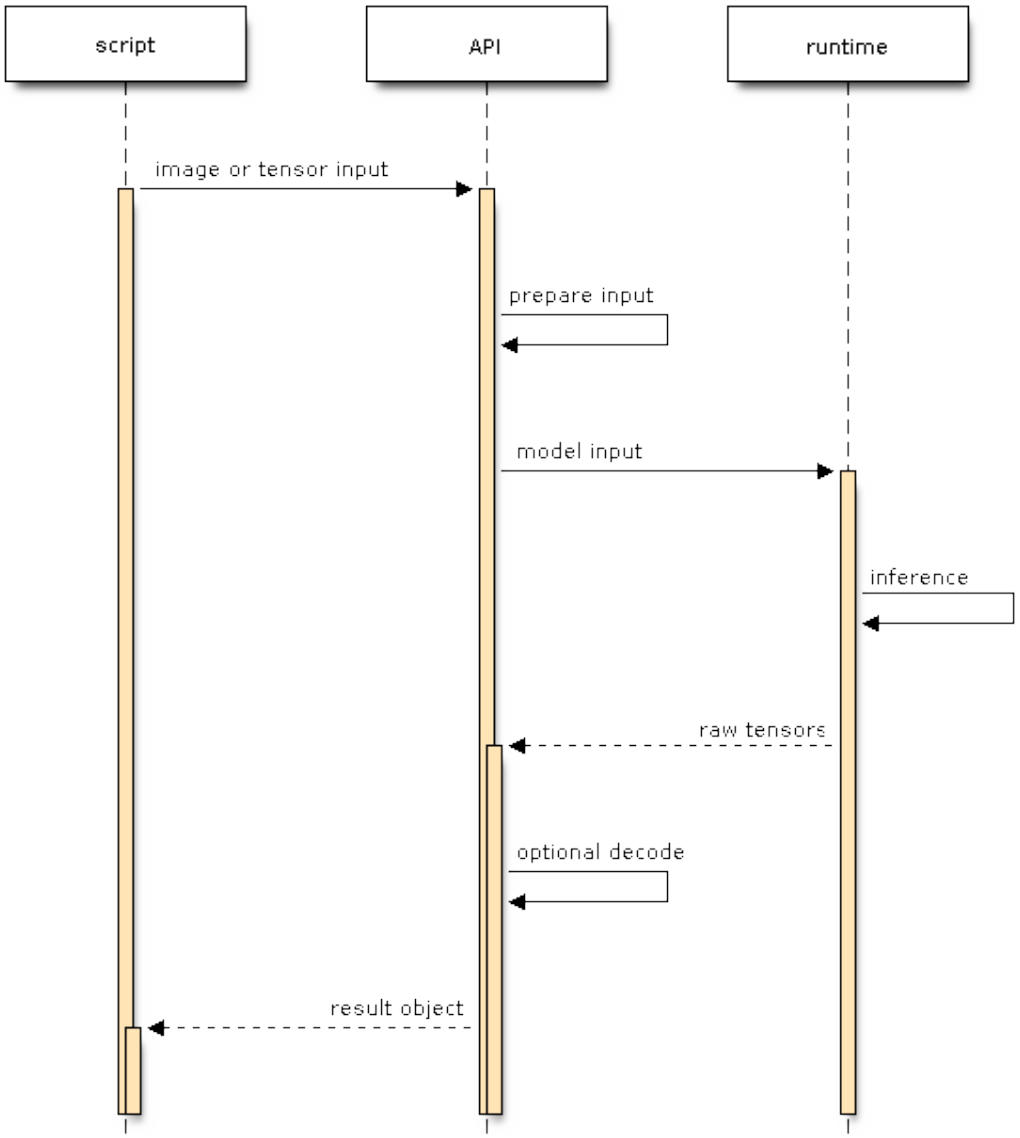
ESP-DL and TFLite Micro are the current model runtime paths exposed by ESP-VISION. The `espdl` module works with `.espdl` files, and `tflite.Model` works with `.tflite` files. Model files are stored on board storage, such as `/sdcard` or `/flash`, and loaded at runtime. The API shape, quantization metadata, and input/output interpretation are defined by the selected runtime and model.

4.5.2 Model Files and Quantization

Models are usually exported or converted offline from a training framework before they are copied to board storage. On microcontrollers, quantized models are preferred because they reduce model size, activation memory, and arithmetic cost. Quantization maps floating-point tensor ranges onto integer values with scale and zero-point metadata; the exact metadata and file layout depend on the runtime and conversion toolchain.

4.5.3 Inference flow

A single inference call usually combines several stages: prepare input data, run the quantized network, and decode raw outputs into values the application can use:



4.5.4 Pre-processing

Before inference the input data must match what the model was trained on:

- **Shape** must match the model input tensor, including resolution and channel count for image models.
- **Color** order and pixel format must match the training pipeline.
- **Normalization or quantization** must use the scale, zero point, mean, standard deviation, or other transform expected by the model. Some APIs expose these as parameters, while examples may show them directly in Python.

4.5.5 Post-processing

Raw network outputs often need task-specific decoding before they are useful to an application:

- **Object detection** (*`espdl.PedestrianDetect`*, *`espdl.ESPDet`*, *`espdl.YOLO11`*) produces candidate boxes with class scores. A confidence `score` threshold drops weak boxes and **non-maximum suppression** (`nms`) removes overlapping duplicates, leaving `(x, y, w, h, score, category)` tuples.

`PedestrianDetect` and `YOLO11` also cap results with `topk`.

- **Pose estimation** (*`espdl.YOLO11nPose`*) adds 17 COCO keypoints per detection.
- **Classification** (*`espdl.ImageNetCls`*) applies an optional `softmax` and returns the `topk (label, score)` pairs.
- **Raw output decoding** (*`espdl.Model`*, *`tflite.Model`*) exposes model outputs before task decoding. Application code must unpack the tensor dtype, use the quantization metadata, and run model-specific steps such as anchor decode, sigmoid or softmax, NMS, top-k selection, and coordinate mapping.

Thresholds and result limits can often be tuned at runtime without reloading the model, which is useful when adapting to lighting, distance, or scene density.

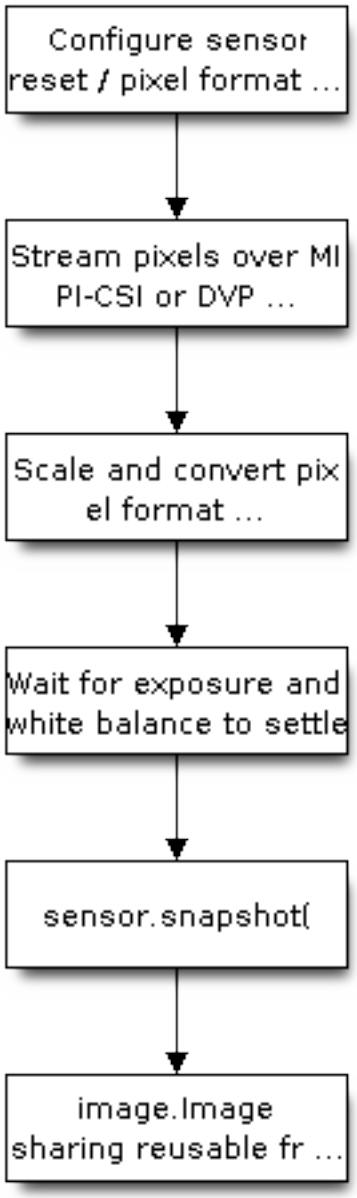
4.5.6 Memory and performance

Models and their activation buffers are large and are allocated in PSRAM. Load a model once and reuse it across frames rather than recreating it per frame. Call `deinit()` (or drop the last reference) to free a model when you are done. Inference cost usually scales with input resolution and model size, so smaller models generally run at higher frame rates than full-size networks.

4.6 The Camera Pipeline

`sensor.snapshot()` looks like a single call, but behind it sits a hardware-specific capture pipeline that turns photons into an *`image.Image`*. Knowing the stages helps explain the configuration calls (`set_pixformat`, `set_framesize`, `skip_frames`) and the performance characteristics of each board.

4.6.1 From sensor to image



The sensor control bus applies the board defaults and requested format before streaming begins. Double buffering allows the next frame to fill while the current frame is processed. ESP32-P4 uses the hardware PPA for scaling and color conversion, while ESP32-S3 performs conversion in software. Calling `sensor.skip_frames(time=2000)` after configuration gives automatic exposure and white balance time to converge.

4.6.2 Per-board backends

The camera backend is selected per board at build time:

Board family	Backend	Notes
ESP32-P4	`esp_video` / V4L2 + PPA	MIPI-CSI sensors; hardware-accelerated scaling and color conversion.
ESP32-S3	`esp32-camera`	DVP sensors; conversion in software, so prefer smaller frame sizes.

Because the public `sensor` API is identical across boards, the same script runs on both; only the achievable resolution and frame rate differ.

4.6.3 Frame sizes

`set_framesize` accepts named sizes (`QQVGA`, `QVGA`, `VGA`, ⋯) drawn from a shared framesize table. Smaller frames cost less memory and less processing time per stage, so when a model or algorithm only needs a small input, capture small rather than capturing large and downscaling.

4.7 Codecs and Streaming

Once you have a frame you often want to *get it off the board*: preview it on a host, record it, or stream it live. ESP-VISION offers several codecs and transports, each suited to a different use case.

4.7.1 Still-image codecs: JPEG and PNG

`image.Image.to_jpeg()` (and its alias *`image.Image.compress()`*) encodes a frame to JPEG. JPEG is lossy: the `quality` keyword (1-100) trades size for fidelity, and `subsampling` controls how aggressively chroma is reduced (`JPEG_SUBSAMPLING_420` is smallest, `444` is best quality). On boards with a hardware JPEG encoder this is offloaded; otherwise a software encoder (`esp_new_jpeg`) is used.

Current firmware does not enable the PNG decoder or encoder. Loading a PNG with `image.Image(path)` raises `OSError` immediately; *`image.Image.to_png()`* and saving an uncompressed image to `.png` raise `RuntimeError`. Convert PNG assets to JPEG before uploading them.

4.7.2 Recording sequences: ImageIO

`imageio.ImageIO` records a *sequence* of frames with their inter-frame timing preserved, so playback runs at the original speed. A **file stream** writes a container on storage; a **memory stream** keeps frames in a pre-allocated PSRAM buffer for fast capture and replay. See *image.ImageIO – Image Stream*.

4.7.3 Video codec: H.264

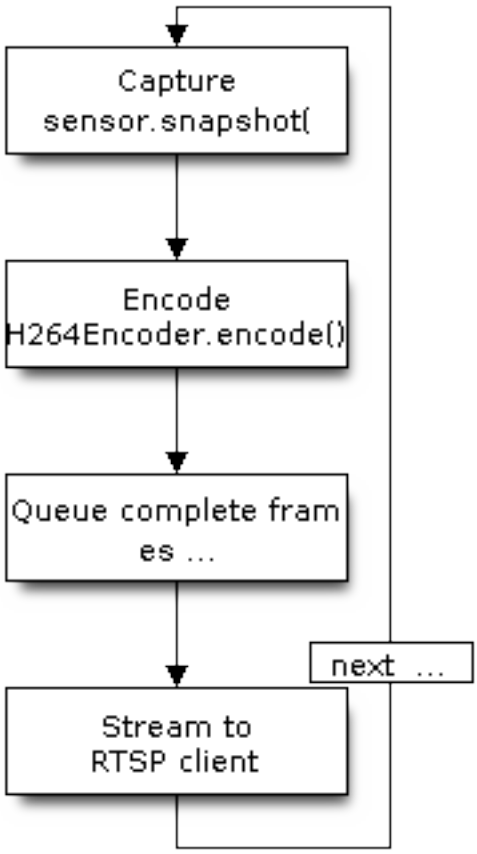
For continuous video, per-frame JPEG is wasteful because it cannot exploit the similarity between consecutive frames. *`h264.H264Encoder`* produces an H.264 stream where occasional **keyframes** (IDR/I) are self-contained and the frames between them only encode what changed. The encoder's parameters tune this trade-off:

- `gop` – keyframe interval. Shorter means faster recovery and easier seeking but larger output; `0` selects one keyframe per second.
- `bitrate` – target bits per second; the dominant size/quality control.
- `qp_min` / `qp_max` – bound the per-frame quantizer.

A lost P-frame corrupts the picture until the next keyframe, which matters for the streaming transport below.

4.7.4 Network streaming: RTSP

rtsp.RTSPServer serves encoded H.264 frames over RTSP so a client such as VLC or ffplay can view the camera at `rtsp://<board-ip>:8554/`. Feed each *h264.H264Encoder.encode()* result to *rtsp.RTSPServer.send()*.



The server keeps a short, in-order frame queue; if a client is slow or absent it drops whole frames rather than blocking the capture loop or sending a partial frame that would corrupt the stream.

4.7.5 Host preview: USB CDC

During development the quickest way to *see* what the camera sees is *image.Image.flush()*, which pushes the current frame to the host as a binary JPEG on the EV-MUX `preview.frame` channel. The companion host tool decodes and displays it. This path needs no Wi-Fi and no SD card, making it the default feedback loop while iterating on a script.

4.8 Startup Sequence

ESP-VISION follows the MicroPython reset and boot model while adding initialization for the Flash filesystem, board storage, camera, display, and preview services. This chapter describes the sequence implemented by the current firmware. For the upstream model and general behavior, see the MicroPython v1.28.0 Reset and Boot Sequence.

4.8.1 Hard Reset and Soft Reset

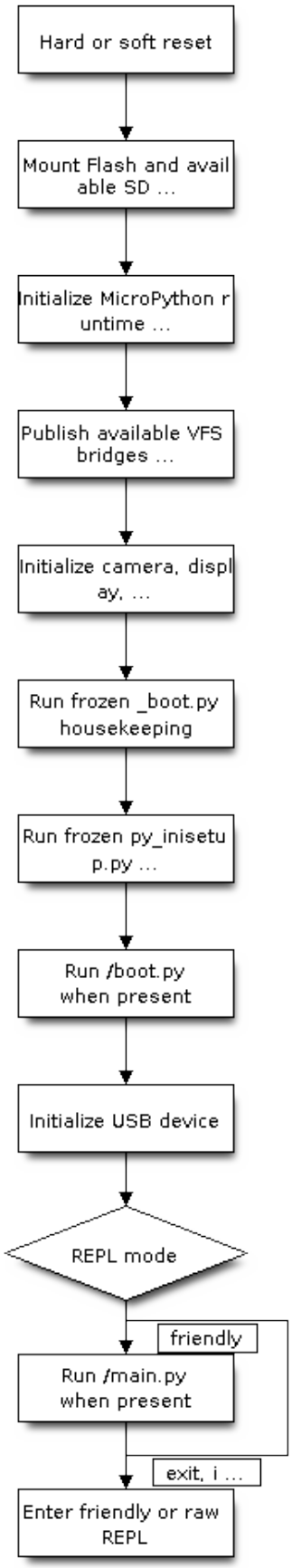
A hard reset restarts the MCU and ESP-IDF runtime before creating a new MicroPython environment. It occurs after power-on, the board reset button, `machine.reset()`, watchdog or brownout reset, and wake-up from deep sleep. Use `machine.reset_cause()` when an application needs to distinguish the reset source.

A soft reset restarts the MicroPython environment without restarting the complete MCU runtime. It can be requested with `Ctrl-D` in the friendly REPL or `machine.soft_reset()`. ESP-VISION clears Python objects and modules, closes files and sockets through MicroPython cleanup, releases camera, display, preview, USB, PWM, timer, UART, thread, and other managed resources, and then repeats the Python startup sequence.

Some system state can survive a soft reset, including the RTC, CPU clock configuration, and an active network interface at the IP layer. Application code must not assume that Python objects representing those resources survive; recreate the objects and verify their state after every reset.

4.8.2 ESP-VISION Startup Order

After a hard or soft reset, ESP-VISION executes the following startup flow:


Hard or soft reset
Mount Flash and avail able SD ...
Initialize MicroPython r untime ...
Publish available VFS bridges ...
Initialize camera, displ ay, ...
Run frozen _boot.py housekeeping
Run frozen py_inisetu p.py ...
Run /boot.py when present
Initialize USB device
REPL mode
friendly
Run /main.py when present
exit, i ...
Enter friendly or raw REPL

The raw REPL used by host automation can skip `main.py` during a soft reset. This allows development tools to gain control without automatically starting the product application.

4.8.3 First Boot and Flash Filesystem

The ESP-IDF storage manager owns the board filesystems independently of the MicroPython VM. By default it attempts to mount the `ffat` partition internally at `/flash` and an available SD card at `/sdcard`. Both volumes are optional and independent. On each VM start, native MicroPython VFS bridges publish the Flash volume at `/` and alias `/flash`, and the SD volume at `/sdcard`. A soft reset recreates these bridge objects without remounting an already mounted physical filesystem; when automatic SD mounting is enabled, an SD card that was unavailable at cold boot is retried on subsequent VM starts. Boards may independently set `ESP_VISION_STORAGE_AUTO_MOUNT_FLASH` or `ESP_VISION_STORAGE_AUTO_MOUNT_SD` to `0` to defer that mount; both default to `1`. See *Storage, Filesystems and Data Protection* for explicit mount retry and fixed USB MSC volume slots.

Automatic initialization uses independent board policies: `ESP_VISION_STORAGE_FLASH_FORMAT_IF_NO_FILESYSTEM` defaults to `1` and `ESP_VISION_STORAGE_SD_FORMAT_IF_NO_FILESYSTEM` defaults to `0`. Exact `FR_NO_FILESYSTEM` initializes internal `ffat` by default and initializes SD only after explicit board opt-in; this can erase factory, foreign or damaged contents. Set the relevant policy to `0` for mount-only automatic startup. Both media still default to automatic mounting, and `storage.retry()` always remains mount-only. A valid FAT is preserved; other mount errors do not trigger automatic formatting or rebooting. Interrupted Flash transactions remain blocked; SD has no equivalent persistent format journal. See *Storage, Filesystems and Data Protection* for policy risks, ownership, backup and confirmed recovery. Status/inspection remain available when files cannot be opened. After explicit recovery, a user-selected soft reset publishes the Python mount. Frozen default-file setup only writes to a usable filesystem and reports errors without crashing.

Existing startup files are not overwritten during a normal firmware update or soft reset. Board packages can provide board-specific default `main.py` and `README.txt` content through `boards/<BOARD>/board_inisetup.py`.

4.8.4 Using boot.py

Use `boot.py` for short, deterministic initialization that must complete before the application starts, such as selecting a product mode, preparing configuration, or bringing up a required network interface:

```
import network

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
```

`boot.py` must return and must not contain the application's permanent loop. ESP-VISION initializes the MicroPython USB device only after `boot.py` completes, so a blocked or long-running `boot.py` can prevent the USB REPL and host tools from becoming available.

4.8.5 Using main.py

Use `main.py` as the product application entry point. Keep the implementation in a separate module so startup policy and application logic remain independent:

```
import sys
import my_app

try:
    my_app.main()
except KeyboardInterrupt:
    raise
except Exception as error:
```

(continues on next page)

(continued from previous page)

```
    print("Fatal application error:")
    sys.print_exception(error)
```

Allowing `KeyboardInterrupt` to propagate lets `Ctrl-C` stop the application and enter the friendly REPL. A production application can instead log the exception and call `machine.reset()` when automatic recovery is required, but an unconditional reset loop can make development and failure diagnosis difficult.

The default ESP-VISION `main.py` prints a board-ready message and sleeps in a loop so the VSCode extension and other host tools can take control. Press `Ctrl-C` to interrupt it and reach the REPL, or replace `/main.py` with the product entry point.

4.8.6 REPL and Recovery

`main.py` exiting normally or raising an uncaught exception transfers control to the friendly REPL. `Ctrl-C` injects `KeyboardInterrupt` into a running Python script, while `Ctrl-D` at the friendly REPL starts a soft reset.

If an application prevents normal startup, connect to the REPL, interrupt it with `Ctrl-C`, and rename or remove the startup file:

```
import os

print(os.listdir("/"))
os.rename("/main.py", "/main.disabled.py")
# Use Ctrl-D to start again without running the previous main.py.
```

If the REPL cannot be recovered, erase and reflash the board from the repository root. `erase-flash` removes the entire device Flash, including the internal filesystem and user data:

```
idf.py --board <BOARD> -p <PORT> erase-flash
idf.py --board <BOARD> -p <PORT> flash monitor
```

Reflashing without `erase-flash` normally preserves the data filesystem and therefore may preserve a faulty `boot.py` or `main.py`.

Chapter 5

Storage, Filesystems and Data Protection

This chapter defines the shared storage contract for firmware, Python programs, Device Core, IDE/Agent clients and recovery tools. Its purpose is to preserve user files and make destructive recovery a user decision. See *storage – Diagnostics and Confirmed Recovery* for the runtime diagnostic and recovery API.

5.1 Media and filesystem ownership

Table 1: Storage responsibilities

Medium	Current use	Contract
Internal Flash `ffat`	FAT, exposed to Python at `/` and alias `/flash`; native IDF path `/flash`; MSC LUN 0 on supported USB-device boards	ESP-IDF owns the filesystem. The Python VFS and MSC share the same bytes, not independent copies. This named data/FAT partition may be initialized by the configured automatic policy or an explicitly confirmed recovery call.
Legacy Flash `vfs`	Mount existing FAT; may contain a filesystem from older firmware	Mount-only in automatic startup and online recovery. Do not reinterpret an unknown or LittleFS volume as disposable FAT.
SD card	FAT at `/sdcard`; MSC LUN 1 on SD-capable USB-device boards	ESP-IDF owns the filesystem. Missing/failed cards do not prevent Flash, the VM or communications from operating. The automatic policy may initialize an unrecognized card; explicit Python SD formatting remains unsupported. Use a selected host card for other recovery after backup.
NVS	Network/settings, writer ownership and format-transaction records	Keep separate from user files. NVS errors do not authorize erasing NVS or formatting a data volume. A Flash-file backup does not include NVS.
External NAND, where fitted	Not currently exposed as an ESP-VISION filesystem, including the onboard NAND on ESP-Mosaico	Report unsupported until a qualified backend exists. Do not treat it as SD or map the raw NOR `ffat` driver onto it; future support must account for NAND bad blocks, ECC and recovery.
Frozen modules / read-only RAM VFS	Firmware modules or temporary model assets	Not a writable user volume or durable backup. RAM data disappears on restart; frozen content belongs to the firmware image.

Board partition tables are authoritative for offsets and sizes. Never infer a destructive range from a module revision, an old example, or a partition with the same label on another board. Board Manager drivers and camera/model consumers must use the common storage owner rather than mount a second filesystem over the same medium.

`/.esp_vision_disk` is only a default-file setup marker. Neither this file nor a saved NVS state is a backup or a new permission to format. Formatting permission comes from the board policy below or an explicit recovery decision.

The Flash backend currently writes through the project's raw block driver with 512-byte logical sectors and 4096-byte erase-block read/modify/write. It is not a journaled filesystem or an ESP-IDF wear-levelling volume. Frequent logs, images and recordings should use an appropriate removable medium and retention limits; account for free space, erase wear and metadata overhead.

5.2 Optional volumes and automatic mounting

Flash and SD are independent optional volumes. By default, startup attempts to mount Flash and, on boards with SD support, an available SD card. A missing/failed Flash volume does not prevent SD from mounting; a missing/failed SD card does not prevent Flash from mounting. Initialization follows the separate policy below; generic mount errors do not trigger it. With neither volume available, the VM and control transports remain usable with a virtual root and diagnostics; startup scripts are not automatically relocated to SD.

Each board may override the following independent defaults in `boards/<BOARD>/boardconfig.h`:

```
#define ESP_VISION_STORAGE_AUTO_MOUNT_FLASH (1)
#define ESP_VISION_STORAGE_AUTO_MOUNT_SD    (1)
#define ESP_VISION_STORAGE_FLASH_FORMAT_IF_NO_FILESYSTEM (1)
#define ESP_VISION_STORAGE_SD_FORMAT_IF_NO_FILESYSTEM    (0)
```

Set either `AUTO_MOUNT` value to `0` to defer that medium's automatic mount. These are independent build-time policies, not persistent Python settings and not hardware capability switches. SD still requires `MICROPY_HW_ENABLE_SDCARD` and the supported SDMMC backend. An otherwise available medium with automatic mounting disabled reports `unmounted` / `STORAGE_NOT_MOUNTED`; `storage.retry("flash")` or `storage.retry("sd")` can explicitly mount it without formatting, even when automatic initialization is enabled. Inspect the result and choose a soft reset to publish a newly mounted Python volume. An SD card that was unavailable at boot is retried on subsequent VM starts only when automatic SD mounting is enabled; already mounted physical filesystems are retained across soft resets.

`ESP_VISION_STORAGE_FLASH_FORMAT_IF_NO_FILESYSTEM` defaults to `1`; `ESP_VISION_STORAGE_SD_FORMAT_IF_NO_FILESYSTEM` defaults to `0`. During automatic mounting, exact FatFs `FR_NO_FILESYSTEM` authorizes formatting and remounting only when that medium's policy is enabled. Internal `ffat` therefore initializes by default, while an unrecognized SD card is preserved unless its policy is explicitly enabled. A nonblank factory image, a foreign filesystem or a damaged FAT may produce this result: it does not prove that the region contains no valuable data. Enabling either policy accepts erasing those contents without runtime confirmation. Keep the relevant policy at `0` where unrecognized data must survive. Legacy `vfs` remains mount-only regardless of either option.

An existing mountable FAT is preserved. `FR_INT_ERR`, I/O/not-ready/write-protection failures, missing media, outstanding handles and denied/quarantined writers do not authorize automatic formatting. Disabled automatic mounts never initialize that medium. `storage.retry()` and writer-handoff refreshes always remain mount-only. Logs identify policy-triggered initialization before destructive writes; failed initialization reports an error without rebooting or crashing.

Internal Flash persists PENDING/READY format transactions in NVS; an interrupted transaction blocks further automatic formatting even when its policy is enabled and needs a new explicit recovery decision. SD currently has no persistent format journal or equivalent interrupted-format guard. If SD automatic initialization is explicitly enabled, a power cut can be followed by another initialization when the next mount returns `FR_NO_FILESYSTEM`. With either policy setting, a partially created FAT may appear readable; that does not prove power-loss recovery. Keep verified off-device backups; the default SD policy preserves unrecognized contents instead of formatting them.

Flash paths `/main.py` and `/flash/main.py` refer to the same file. The alias is a second Python VFS bridge to the existing IDF filesystem, not another FAT mount, copy or USB disk. Relative paths follow the selected bridge's current directory. Use one consistent prefix for both arguments of `os.rename()`: MicroPython rejects a rename between distinct VFS mount points, including `/` and `/flash`. `/flash` is reserved as a mount point; a pre-existing

physical `flash` directory is retained and can be accessed at `/flash/flash`. `/flash/sdcard` refers to a directory on Flash, not the separate SD volume at `/sdcard`.

Backup and restore tools must identify volumes rather than count path prefixes: do not back up `/` and `/flash` as separate copies or recursively traverse synthetic mount points. Prefer `/flash` as the explicit Flash backup root, and back up `/sdcard` separately.

5.3 USB with two volumes

Default labels are `ESP_FLASH` and `ESP_SDCARD`. Automatic mounts name only unlabeled volumes and preserve custom labels; label errors never authorize formatting. See *USB Interfaces, Configuration and Disk Names* for USB configuration, naming limits, interface strings and host drive letters.

On boards/ports with native USB device MSC enabled, one MSC interface exports fixed logical-unit slots: LUN 0 is internal Flash and LUN 1 is SD when the board supports it. If both volumes are mounted, healthy and admitted to the host by the writer gate, the host can enumerate two separate media. Drive letters, mount paths and empty-slot presentation are controlled by the host OS. A board without SD support exports only the Flash slot. The USB Serial/JTAG debug interface does not provide these MSC disks.

An absent, unmounted or unavailable medium reports not ready for that slot; SD never shifts to LUN 0 when Flash is unavailable. Turning off automatic mounting defers that slot's readiness, not the other volume or the entire USB interface. `/flash` creates no additional LUN. Ejection and writer handoff are per volume: safely eject the selected OS volume and verify its ownership before Python/IDE writes; two visible disks do not authorize simultaneous host and VM writes to either shared filesystem.

5.4 Failure policy: preserve data and keep control available

A mount, allocation, NVS, write, close, sync or card-readiness error is a recoverable storage condition. Storage error paths must return status/errors, close admission when necessary, and keep the REPL, session management and network/control transports available. They must not call a fatal assertion, abort, crash or reboot to hide the failure. Python argument exceptions and an operation returning `ok: false` are not a device crash.

Mount failure alone is not formatting permission: the automatic exception is the per-medium board policy for exact `FR_NO_FILESYSTEM` described above, enabled for Flash and disabled for SD by default. When the relevant policy is disabled, blank, factory/unknown and damaged volumes remain untouched. Missing NVS records never independently authorize formatting, and NVS errors do not bypass the Flash transaction guard. `storage.retry()` never creates a filesystem, erases a region or resets the device.

Table 2: Diagnosis and choices

State	Meaning	User-directed next step
`ready`	Filesystem mounted; writer permission remains a separate condition	Use files under the writer rules and maintain a verified off-device backup.
`blank`	The whole partition was read and found erased	Confirm initialization if this is the intended empty volume.
`unmounted`	Automatic mounting is disabled for this otherwise available medium	Use `storage.retry()` if mounting is wanted; this is not a format request.
`unrecognized`	No mountable FAT; contents may be factory assets, another filesystem or damaged user data	Preserve/export required data, inspect the layout, then choose retry, offline recovery or confirmed formatting.
`absent / unsupported`	Medium missing or operation unavailable	Check the card/board capability. Do not propose formatting absent hardware.
`io_error / unavailable`	Driver, mount or resource failure	Inspect the error code, power, connection and free resources; retry only when the cause is addressed.
`state_error`	NVS format-transaction metadata cannot be read or validated	Preserve NVS and the data volume for diagnosis; do not clear history to bypass the guard.
`format_interrupted`	Formatting failed before completion	Flash persists this guard and requires a new decision; SD has no equivalent persistent journal. Inspect the medium and the board policy before restarting.
Writer held / quarantined	Another writer owns the medium, or a handoff outcome is uncertain	Finish the verified handoff or investigate quarantine. A format request does not override ownership.

`storage.status()` reports the state, stable code, human-readable message, mount/presence/capability flags, native error, FatFs result when known, and Flash geometry. Device Core `inspect({ fields: ['storage'] })` retains the writer fields and also returns `fs_state`, `fs_error`, `fatfs_result` and `format_supported`. These fields describe a snapshot, not permission to destroy data. Logs retain actionable storage messages after transport startup, and native uploads to an unmounted volume return `STORAGE_UNAVAILABLE` with recovery guidance instead of an opaque file-open failure.

A frontend must show the affected volume, reason and data impact; offer inspect/retry/export before an explicit destructive choice; and leave manual formatting unselected until a deliberate user confirmation. Agents and unattended jobs must not infer new runtime confirmation from a timeout, reconnect, generic request to fix an error, or factory data. The firmware ' s configured automatic initialization is a separate boot policy. This change supplies status, logs, errors and an explicit REPL API; it does not claim that every existing IDE/Web UI already has a backup wizard or format button.

5.5 One writer, safe publication and power loss

EV-MUX session ownership is not filesystem write ownership. Before Python/IDE writes, backup, restore or unmount, coordinate the volume with the OS MSC host and the storage-writer handoff described in *Interactive Python, scripts and file operations*. Eject the OS volume, stop application/background writers, drain handles and writes, and check the actual writer state. An uncertain handoff remains quarantined across reboot. Legacy compatibility mode is not a qualified concurrent-writer guarantee; merely connecting the SDK does not make simultaneous OS and Python writes safe.

Unmounted Flash is not ready for MSC reads or writes and is not published until its format transaction completes. This transaction guard does not preserve future files if FAT later becomes unrecognizable while the automatic initialization policy remains enabled. SD readiness is independent and its format has no persistent transaction guard. Do not bypass a failed close, outstanding upload descriptor or writer quarantine to make a format succeed.

Native binary upload uses a temporary file, expected length/SHA-256, sync/close and final publication. Ordinary replacement failure attempts rollback from an old-file backup. This protects against transport errors and ordinary

publication errors, not every power cut between FAT directory updates. The staging/rollback file on the same medium is not an independent backup. `fsync` and a successful transfer response do not turn FAT or the raw Flash erase-block update into a journaled, power-loss-atomic transaction. After a lost final response, inspect the destination before retrying.

Write important application data to a new file, validate and close/sync it, and keep an earlier verified generation until publication is checked. Use bounded logs and reserve free space for both the new and old generation. If a write or close fails, stop further writes and report the original error; repeatedly retrying or deleting unrelated files is not recovery.

5.6 Backup and restore policy

A backup must live on a different failure domain: normally the host computer or another independently retained medium. A directory or renamed old file on the same Flash/card does not protect against corruption, accidental formatting or media failure.

1. Identify the physical device, board, firmware and selected volume. Stop scripts and every background writer; safely eject MSC and verify ownership before taking a filesystem-consistent copy. If writers cannot be stopped, label the result as an unverified copy rather than a consistent backup.
2. Export user scripts, hidden configuration, required packages, model sidecars and other required files. Record paths, sizes and SHA-256 hashes in a host-side manifest along with device identity and filesystem/layout metadata. Preserve previous backup generations instead of overwriting the last known good copy.
3. Verify the downloaded bytes against the manifest and check that expected files are present. A completed download alone is not proof of a usable backup. Periodically test restoration on a suitable spare medium/device.
4. Restore only after verifying the target identity, compatible layout and enough free space. Use the normal verified upload/ownership path, inspect length/hash/readback, and retain the backup until validation succeeds. Do not automatically execute restored startup scripts before the user has checked them.

Current file APIs allow selected-file export/import through Device Core or host copying after a safe MSC handoff. A complete manifest-based backup/restore wizard is not implemented by this firmware API. Hosts implementing one must follow the rules above and distinguish successful, partial, failed and unverified copies; they must never call a partial copy a verified backup.

If the filesystem cannot mount, a selected raw partition image can preserve evidence for offline recovery. Entering ROM mode interrupts runtime; a user-requested read may use the actual offset and length from the installed partition table:

```
esptool --chip <CHIP> --port <PORT> read-flash <OFFSET> <LENGTH> <HOST_BACKUP.bin>
```

A raw image is not a verified file-level backup and may require the matching layout and encryption context to be usable. NVS/system-state backups are a separate explicit scope and may include credentials. Protect those backup files appropriately. Routine development flashing does not automatically back up/restore firmware or read the whole chip; such operations run only when the user requests them.

5.7 Explicit recovery of an unmounted Flash volume

Inspect without writing:

```
import storage
print(storage.status("flash"))
print(storage.status("sd"))
print(storage.retry("flash"))  # mount retry only; never formats
```

If `retry` returns `ok: True` and `soft_reset_required: True`, the user can issue a soft reset to publish the new Python mount. Failures leave the runtime available and retain the diagnostic state.

Only after the user has selected the correct Flash volume, preserved any needed files and agreed to erase it, run the following separate destructive operation:

```
result = storage.format("flash", confirm=True)
print(result)
```

The literal boolean `True` is required; omitted/false/truthy-string confirmation is rejected. Recovery only formats an unmounted data/FAT `ffat` after open handles and writer admission are checked. It refuses a mounted/readable volume, legacy `vfs`, online SD formatting and an unavailable/quarantined writer. Hardware I/O failures do not trigger formatting. The range comes from the device' s partition table; NVS, firmware and SD are not erased.

Before formatting, a partition-bound NVS `PENDING` record is committed. Completion requires successful format, remount and a durable `READY` record before publishing the volume. A failed commit, format or remount leaves the volume unavailable; an interrupted `PENDING` state never resumes on its own, even if a partial FAT header looks readable. A new explicit user decision is required. Clearing NVS is not an approved way to bypass this state.

Check `ok` and `operation_error`. If success requests `soft_reset_required`, explicitly soft reset to publish the Python mount and create missing default files:

```
import machine
machine.soft_reset()
```

A reset is a user-selected follow-up, never an automatic response to a storage fault. Reinspect storage and verify normal file creation, upload, readback and persistence across restart before restoring the application. A failure or lost response has uncertain completion; inspect state before repeating a destructive operation.

5.8 Firmware installation and acceptance

Ordinary firmware updates preserve user files and NVS. Full installation/layout changes and whole-chip erase have different data effects; hosts must describe those effects before the user selects them. Never silently switch an update to an installation or format a volume because its layout differs. See *Application API reference* for the update/install contract.

Acceptance must cover blank and residue-filled media, valid existing FAT, damaged FAT, missing/removed SD, NVS failures, full media, write/close/sync faults, interrupted formatting and uncertain writer handoff. Verify zero automatic erase/format/NVS writes for mount failures outside the enabled, exact `FR_NO_FILESYSTEM` initialization policy. For policy-authorized initialization, verify its medium-specific defaults and Flash PENDING/READY transaction guards. Also verify continued session/REPL operation, useful status/errors, explicit confirmation guards, and isolation of unrelated volumes. Host fault-injection tests do not establish real power-loss or affected-module qualification; report the exact commit, medium, board, interface and actual hardware evidence.

Chapter 6

API Reference

ESP-VISION provides Python APIs for camera capture, image processing, display, video encoding and streaming, and AI inference. Developers can also directly use the standard MicroPython APIs enabled in the firmware for hardware control, networking, file access, system services, and general application development. ESP-VISION maintains compatibility with commonly used OpenMV module names where applicable, including `sensor` and `image`, and additionally provides modules such as `display`, `espdl`, and `tflite`.

The ESP-VISION module pages are kept in sync with the type stubs under `stubs/`, which are also usable for IDE completion. Their availability is derived from `micropython.cmake` during the documentation build. The *MicroPython APIs* page indexes the language, standard-library, hardware, networking, filesystem, and runtime APIs that ESP-VISION firmware inherits from the pinned MicroPython version.

6.1 MicroPython APIs

ESP-VISION is built on the MicroPython ESP32 port and directly retains its language, standard-library, hardware-control, networking, filesystem, and runtime APIs. These APIs can be used together with `sensor`, `image`, `display`, `espdl`, and the other ESP-VISION modules in the same script; ESP-VISION does not wrap or rename them.

6.1.1 Documentation Baseline

ESP-VISION pins MicroPython v1.28.0. Use the MicroPython v1.28.0 documentation as the authoritative API reference, especially the MicroPython libraries and ESP32 quick reference. MicroPython implements a resource-oriented subset of CPython, so CPython documentation must not be treated as an exact description of module contents or behavior.

6.1.2 Language and Built-in APIs

ESP-VISION retains the MicroPython language and built-in types, functions, exceptions, iteration protocol, context managers, classes, and asynchronous syntax. See MicroPython language and implementation and builtins for details.

Use exceptions and context managers to keep long-running vision applications recoverable and to close files deterministically:

```
try:
    with open("/sdcard/result.txt", "w") as output:
        output.write("capture complete\n")
```

(continues on next page)

(continued from previous page)

```
except OSError as error:
    print("storage error:", error)
```

6.1.3 Standard and Micro Libraries

Common inherited modules include `array`, `asyncio`, `binascii`, `cmath`, `collections`, `errno`, `gc`, `gzip`, `hashlib`, `heapq`, `io`, `json`, `marshal`, `math`, `os`, `platform`, `random`, `re`, `select`, `socket`, `struct`, `sys`, `time`, `weakref`, `zlib`, and `_thread`. ESP-VISION board manifests also freeze MicroPython's `asyncio` package into the firmware.

Refer to the MicroPython library index for each module's classes and functions. A documented upstream module may still be omitted or reduced by the selected firmware configuration.

6.1.4 JSON and Time Example

```
import json, time

result = {
    "timestamp_ms": time.ticks_ms(),
    "objects": 3,
    "confidence": 0.91,
}
payload = json.dumps(result)
print(payload)
```

`time.ticks_ms()` is suitable for measuring intervals and generating local runtime timestamps; use `time.ticks_diff()` when subtracting tick values so wraparound is handled correctly. `json` is useful for serializing inference results before storage or network transmission.

6.1.5 Hardware and ESP32 APIs

The inherited machine module provides MCU control and peripheral classes including `Pin`, `Signal`, `ADC`, `ADCBlock`, `PWM`, `I2C`, `SoftI2C`, `SPI`, `SoftSPI`, `UART`, `Timer`, `RTC`, `WDT`, `SDCard`, and, where supported, `DAC`, `I2S`, and `I2CTarget`. Pin assignments and peripheral conflicts remain board-specific, so consult the board schematic before combining these APIs with camera, display, storage, or codec hardware.

The inherited esp and esp32 modules expose ESP32-port-specific system and peripheral functions. Availability of individual classes such as `esp32.PCNT`, `esp32.RMT`, and other SoC-specific blocks depends on the selected chip and ESP-IDF version.

6.1.6 GPIO and PWM Example

```
from machine import Pin, PWM

led = Pin(LED_GPIO, Pin.OUT)
led.value(1)

pwm = PWM(Pin(PWM_GPIO), freq=1000, duty_u16=32768)
pwm.duty_u16(49152)
pwm.deinit()
```

Replace `LED_GPIO` and `PWM_GPIO` with pins that are free on the selected board. Check the schematic and board configuration first because camera, display, storage, USB, and codec peripherals may already reserve GPIOs.

6.1.7 Networking APIs

The inherited network, socket, and select modules provide network-interface management and TCP/UDP communication. `network.WLAN` is enabled in the current product board configurations, but successful use still requires compatible networking hardware and firmware configuration.

The current board profiles disable bluetooth and espnow. SSL/TLS, `cryptolib`, WebSocket and socket-event support depend on the ESP-IDF version used to build the firmware. Traditional MicroPython WebREPL is not supported by the ESP-VISION EV-MUX runtime, even when `_webrepl` is compiled; its terminal and file operations do not use the EV-MUX owner authorization. Use *Application API reference* over TCP for network control, and see *Chip and Board Support* for the maintained support summary.

6.1.8 Connect to Wi-Fi

```
import network, time

wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect("ssid", "password")

for _ in range(100):
    if wlan.isconnected():
        break
    time.sleep_ms(100)

if not wlan.isconnected():
    raise OSError("Wi-Fi connection failed")
print("address:", wlan.ifconfig()[0])
```

Wi-Fi requires compatible hardware and board configuration. Production code should apply a timeout, avoid printing credentials, and handle reconnects rather than waiting indefinitely.

6.1.9 Filesystem and Runtime APIs

ESP-VISION retains MicroPython virtual-filesystem and stream behavior, including os, io, vfs, and the built-in `open()` function. Board startup code mounts product storage such as `/flash` and `/sdcard` when the corresponding partition and hardware are available.

Runtime services remain available through micropython, gc, `sys`, `time`, and `_thread`. ESP-VISION enables MicroPython threading with a GIL; vision objects and hardware resources should still be treated as shared resources unless an API explicitly documents concurrent use.

6.1.10 List Storage and Check Memory

```
import gc, os

print("flash:", os.listdir("/flash"))
if "sdcard" in os.listdir("/"):
    print("sdcard:", os.listdir("/sdcard"))

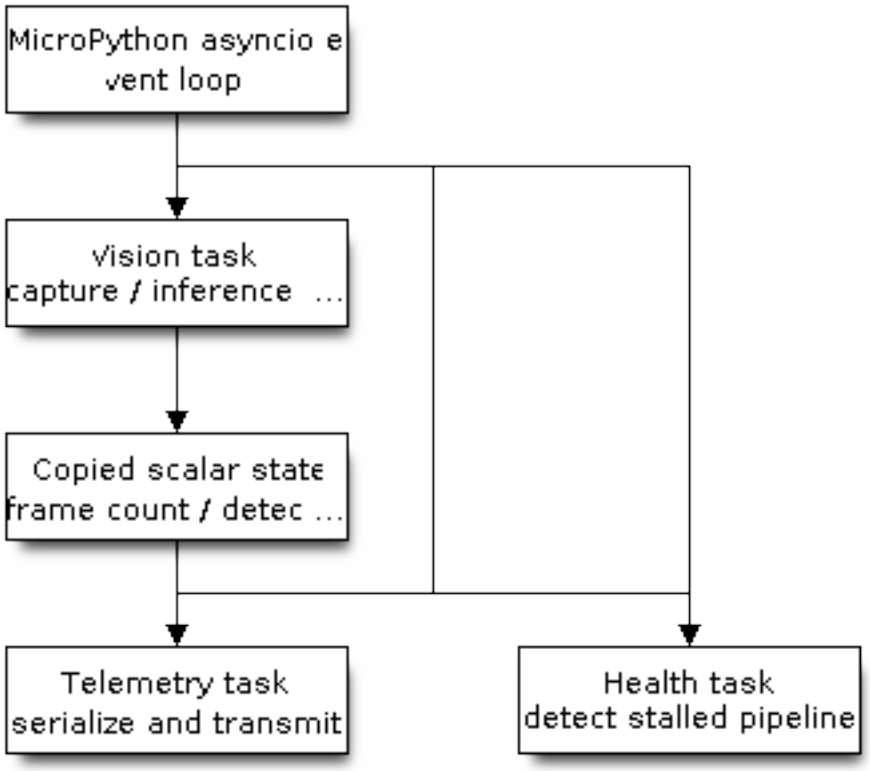
gc.collect()
print("free heap:", gc.mem_free())
```

Check for optional mount points before using them. `gc.collect()` can be useful before loading a large model or allocating a frame history, but repeatedly forcing collection in every frame loop may reduce throughput.

6.1.11 Structuring a Complex Application

For developers familiar with ESP-IDF, an `asyncio` task is a cooperatively scheduled Python coroutine, not a FreeRTOS task. Multiple coroutines share one Python execution context and switch only when they `await`. This is the preferred structure for control logic, network I/O, status reporting, and time-based work because it avoids concurrent access to camera, model, display, and frame-buffer objects.

The following structure assigns exclusive ownership of the vision pipeline to one coroutine while telemetry and health-monitoring coroutines consume only copied scalar results:




```
import asyncio
import espdl
import json
import sensor
import time

MODEL = "/sdcard/espdet_pico_224_224_face.espdl"

state = {
    "frames": 0,
    "detections": 0,
    "last_frame_ms": time.ticks_ms(),
}
state_lock = asyncio.Lock()

async def vision_task(detector):
    while True:
        img = sensor.snapshot()
        results = detector.detect(img)
```

(continues on next page)

(continued from previous page)

```
        for x, y, w, h, score, category in results:
            img.draw_rectangle(x, y, w, h, color=(255, 0, 0))
        img.flush()

        async with state_lock:
            state["frames"] += 1
            state["detections"] = len(results)
            state["last_frame_ms"] = time.ticks_ms()

        # Let ready control and network tasks run between frames.
        await asyncio.sleep_ms(0)

async def telemetry_task():
    while True:
        await asyncio.sleep_ms(1000)
        async with state_lock:
            payload = json.dumps({
                "frames": state["frames"],
                "detections": state["detections"],
            })
        print("telemetry:", payload)
        # Replace print() with non-blocking socket transmission.

async def health_task():
    while True:
        await asyncio.sleep_ms(500)
        async with state_lock:
            age = time.ticks_diff(
                time.ticks_ms(), state["last_frame_ms"]
            )
        if age > 3000:
            print("warning: vision pipeline stalled")

async def main():
    sensor.reset()
    sensor.set_pixformat(sensor.RGB565)
    sensor.set_framesize(sensor.QVGA)
    sensor.skip_frames(time=1000)
    detector = espdl.ESPDet(MODEL, score=0.5, nms=0.7)

    vision = asyncio.create_task(vision_task(detector))
    telemetry = asyncio.create_task(telemetry_task())
    health = asyncio.create_task(health_task())
    try:
        await asyncio.gather(vision, telemetry, health)
    finally:
        vision.cancel()
        telemetry.cancel()
        health.cancel()
        detector.deinit()

asyncio.run(main())
```

Calls such as `sensor.snapshot()`, inference, image processing, and `img.flush()` are synchronous. While one of these calls is running, other coroutines cannot run; the explicit `await asyncio.sleep_ms(0)` provides a scheduling point between frames. If a network operation can block, use an `asyncio` stream or non-blocking socket rather than a long blocking call.

6.1.12 Threads and Native Tasks

The inherited `_thread.start_new_thread()` API creates another FreeRTOS task, but it is not equivalent to `xTaskCreatePinnedToCore()` as an application interface. Python does not expose task priority or core affinity here; MicroPython threads are pinned to `MP_TASK_COREID` and share the interpreter through the GIL. `_thread.stack_size()` can configure the stack allocated for subsequently created threads.

Use `_thread` only when a blocking operation cannot be integrated with `asyncio`, protect shared Python data with `_thread.allocate_lock()`, and keep camera, display, codec, model, and frame-buffer ownership in one thread unless an API explicitly documents thread safety. An IRQ or another thread can notify the event loop with `asyncio.ThreadSafeFlag` without directly manipulating vision objects.

For deterministic latency, explicit FreeRTOS priorities or core affinity, ISR-to-task notification, or continuous work that must proceed while Python holds the GIL, implement the worker as an ESP-IDF C/C++ component and expose a narrow Python API. In ESP-IDF terms, use `asyncio` tasks for application state machines, `_thread` for exceptional blocking integration, and native tasks for real-time services.

6.1.13 Check Availability on a Firmware

Standard MicroPython API availability is controlled by `overlay/micropython/ports/esp32/mpconfigport.h`, the selected board's `mpconfigboard.h` and `mpconfigboard.cmake`, ESP-IDF version checks, and SoC capability macros. Because availability is not determined solely by the chip, the definitive check is the exact firmware:

```
help("modules")
import machine
help(machine)
```

Use `hasattr()` or a guarded import when an application must run across multiple ESP-VISION boards. To remove or add inherited APIs in a product firmware, follow *Customize Firmware Features*.

See also:

Runnable examples that use standard MicroPython APIs include `example/00-HelloWorld/helloworld.py` (first script) and `example/06-Peripherals/00-Storage/sdcard.py` (requires SD media). `example/07-Network/00-WebREPL/webrepl.py` documents the unsupported traditional WebREPL boundary and exits before network changes; it is not a runnable network terminal on ESP-VISION.

6.2 sensor –Camera

The `sensor` module controls the camera and captures frames. It mirrors the OpenMV `sensor` API so existing OpenMV scripts port with little change.

6.2.1 Camera Initialization and Continuous Capture

A typical program resets the sensor, selects a pixel format and frame size, lets automatic exposure and white balance settle, and then captures frames continuously:

```
import sensor

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.skip_frames(time=2000)
```

(continues on next page)

(continued from previous page)

```
while True:
    img = sensor.snapshot()
```

`sensor.reset()` applies the board's default camera configuration. `RGB565` is suitable for display, drawing, and most AI workflows, while `GRAYSCALE` reduces memory and processing cost for many detection algorithms. `sensor.snapshot()` returns an *image.Image* backed by the reusable frame buffer, so copy the image when it must remain unchanged after the next capture. You can pass a writable buffer large enough for the configured frame as `sensor.snapshot(buffer)`; the returned image keeps that buffer alive.

6.2.2 Image Orientation and Camera Status

Use horizontal mirror and vertical flip to correct the image orientation imposed by the physical sensor installation. `status()` provides the active dimensions, pixel format, sensor ID, orientation, and crop information for diagnostics:

```
import sensor

sensor.reset()
sensor.set_hmirror(True)
sensor.set_vflip(False)

info = sensor.status()
print("sensor id:", info["id"])
print("output:", info["width"], "x", info["height"])
print("ready:", info["ready"])
```

Mirror and flip settings affect subsequently captured frames. Product code can call `status()` after initialization to verify that the camera is ready and that the negotiated output size matches the processing pipeline.

6.2.3 Temporarily Stop Capture

The camera stream can be stopped when capture is not required and restarted before the next frame:

```
sensor.shutdown(True)
# Perform work that does not require the camera.
sensor.shutdown(False)
sensor.skip_frames(n=3)
img = sensor.snapshot()
```

After restarting the stream, discard a few frames if exposure or the incoming frame queue needs to stabilize.

Note: The default `ESP32_P4X_EYE` SPI settings support QVGA camera capture and LCD drawing at the same time. SPI color transfers use DMA-capable memory, so raising the camera frame size or the panel transfer size/queue depth can exceed the available contiguous DMA block even when `gc.mem_free()` and total free PSRAM remain large. If `display.Display()` or `lcd.write()` raises `ESP_ERR_NO_MEM`, drop any `sensor.snapshot()` image references, call `sensor.shutdown()`, initialize and use the LCD, then call `lcd.deinit()` followed by `sensor.reset()` to resume camera capture.

6.2.4 IPA Runtime Control

IPA and ISP control methods are optional firmware features. Check `hasattr(sensor, "get_ipa")` or `hasattr(sensor, "get_isp")` before using them; these methods are absent when the feature is not built in.

Use `sensor.set_ipa(False)` to stop the image processing algorithm (IPA) controller and `sensor.set_ipa(True)` to start it again. Initialize the camera before using these methods. `sensor.get_ipa()`

returns the controller's current state, or `None` when runtime control is unsupported. Setting IPA on an unsupported camera raises `OSError`.

```
sensor.reset()
sensor.skip_frames(time=2000)
if hasattr(sensor, "get_ipa") and sensor.get_ipa() is not None:
    sensor.set_ipa(False)
    print("IPA:", sensor.get_ipa())
    img = sensor.snapshot()
    img.flush()
    sensor.set_ipa(True)
```

Changing IPA state briefly stops and restarts capture while preserving the output format, frame size, mirror, and flip settings. Setting the current state again has no effect. `sensor.reset()` and restarting after `sensor.shutdown(True)` restore the default IPA state.

A failed switch leaves the camera shut down rather than in its previous state: the capture buffers, the scaler, and the camera device itself are all released before `OSError` is raised. Call `sensor.reset()` to bring the camera back.

Disabling IPA stops its automatic parameter updates; it does not bypass the ISP or clear existing tuning. The image may initially look unchanged. To compare the effect, change the illumination or light source color while previewing, then enable IPA and allow its algorithms to settle again. This switch controls only algorithms declared in the active IPA JSON configuration and supported by the backend. Enabling IPA returns its configured blocks, such as CCM, gamma, and sharpening, to automatic control; manual values are not preserved as overrides. This switch does not disable automatic exposure or white balance implemented inside the image sensor. On ESP32_P4X_EYE, the OV2710 controls exposure and gain automatically inside the sensor, independently of IPA; stopping IPA neither freezes these parameters nor makes manual exposure or gain controls available.

6.2.5 Reading IPA Statistics

`sensor.get_ipa_stats(timeout=1000)` collects one new ISP statistics packet while IPA is running. Check `hasattr(sensor, "get_ipa_stats")` before use. Initialize the camera and enable IPA first. `timeout` is an integer from 1 through 1000 milliseconds; invalid values raise `ValueError`. The call returns `None` on timeout or when runtime IPA control is unsupported, and raises `OSError` for an uninitialized camera, disabled IPA, or a device error.

Each read creates a one-entry statistics queue and releases it before returning, including on timeout. It does not stop capture or alter tuning. Reads briefly add copying work to the controller and may block Python for the timeout duration, so sample periodically rather than on every preview frame. Keep calling `sensor.snapshot()` between reads to drain camera capture buffers. The statistics sequence is independent of the Python image returned by `snapshot()`; the two are not guaranteed to describe the same frame.

The returned dictionary always contains these keys. A statistics block without a valid flag in this packet is `None`; blocks are not accumulated across packets.

Table 1: IPA statistics

Key	Value
`seq`	64-bit statistics sequence; may restart after camera/IPA initialization.
`flags`	IPA validity bits: AWB=1, AE=2, HIST=4, SHARPEN=8, AF=16, AWB_SUBWIN=32. These differ from the raw ESP Video AE/AWB bits.
`ae`	5×5 luminance grid indexed as `[x][y]`.
`awb`	Dictionary with `counted` (white patch count), `sum_r`, `sum_g`, and `sum_b`. A valid result can have zero white patches.
`awb_subwin`	5×5 grid indexed as `[x][y]`, each cell with the same fields as `awb`. Requires backend, chip, and statistics configuration support.
`hist`	16 pixel counts in hardware histogram segment order; segment boundaries depend on the active ISP configuration.
`sharpen`	Maximum high-frequency pixel value.
`af`	Three window dictionaries, each with `definition` and `luminance`. These are focus statistics, not lens positions.

```
sensor.reset()
sensor.set_ipa(True)
sensor.skip_frames(time=2000)
sensor.snapshot()
stats = sensor.get_ipa_stats(timeout=200)
if stats is not None:
    print(stats["seq"], stats["ae"], stats["awb"])
```

For a valid `awb` block with `counted > 0`, dividing each channel sum by `counted` yields that statistics source's channel mean. Hardware white-patch statistics are filtered samples, not full-frame RGB means; the OV2710 backend can instead provide the sensor's before-gain channel averages with `counted=1`. Neither source is a mean of the rendered `image.Image`.

See `example/01-Camera/04-IPA/ipa_stats.py` for periodic statistics reads during preview. This API exposes the ISP inputs consumed by IPA; it does not report inferred color temperature, converged exposure, or other algorithm outputs. Use `get_isp()` for supported driver parameter readback.

6.2.6 Manual ISP Control

After initialization, `sensor.get_isp(block)` returns a dictionary of the current driver parameters, or `None` when the block is unsupported. A block is unsupported when the camera driver has no getter for it or when the firmware was built for a chip revision that lacks the hardware: `lsc` needs ESP32-P4 v1.0 or newer and `blc` needs v3.0 or newer, while `exposure` is write-only on sensors such as SC2336, so `sensor.get_isp("exposure")` normally returns `None`. The OV2710 driver used by ESP32_P4X_EYE implements neither manual `exposure` nor `pixel_gain`: reads return `None` and writes raise `OSError` with `ESP_ERR_NOT_SUPPORTED`. A write-only getter on another sensor does not imply that its setter is unsupported. Call `sensor.set_ipa(False)` before writing manual settings. `sensor.set_isp(block, **params)` updates one block, keeping unspecified fields from its current configuration. Writes while IPA is running raise `OSError` to prevent the algorithms from overwriting manual settings.

```
sensor.reset()
sensor.skip_frames(time=2000)
sensor.set_ipa(False)

print(sensor.get_isp("wb"))
sensor.set_isp("wb", enable=False)
sensor.set_isp("ccm", enable=False)
sensor.set_isp("gamma", enable=False)
sensor.snapshot().flush()
```

(continues on next page)

(continued from previous page)

```
sensor.set_isp("wb", enable=True, rg=1.8, bg=1.4)
# On sensors with manual exposure and gain support (not OV2710):
# sensor.set_isp("exposure", value=10000)  # Microseconds, rounded to 100 us.
# sensor.set_isp("pixel_gain", value=2.0)  # Gain multiplier.

sensor.set_ipa(True)
```

`get_isp()` reads driver configuration, not a hardware commit acknowledgement or a measurement of the returned image. In particular, `get_isp("wb")` reports the ISP white balance stage; it does not report the OV2710 sensor's internal AWB gains. `enable=False, rg=1.0, bg=1.0` can therefore coexist with sensor-side AWB. On P4 revisions without a separate white balance stage, the driver combines manual white balance gains with CCM.

Capture buffers may already contain frames produced before a write. Sleeping does not consume those frames. Continue capturing, or call `sensor.skip_frames(n=3)` to consume queued frames before evaluating a change. This discards older frames; it is not a hardware commit barrier or a guarantee that automatic sensor controls have settled. A repeated write or a fixed sleep is not a reliable way to establish which configuration produced a frame.

The supported block names and fields follow the tuning parameter vocabulary:

Table 2: ISP parameters

Block	Fields
`exposure`	`value`: exposure in microseconds, positive. The device control counts 100-microsecond units, so a write is rounded to the nearest 100 microseconds and must then fit the range and step the sensor reports, which is where the usable limits come from. SC2336 at 1280x720 30fps, for example, accepts roughly 200 microseconds through 33.2 milliseconds in 100-microsecond steps.
`pixel_gain`	`value`: positive gain multiplier, selected from the nearest supported gain menu entry within the sensor's range.
`demosaic`	`enable`, `gradient_ratio`: nonnegative gradient ratio within the driver's fixed-point range.
`wb`	`enable`, `rg`, `bg`: red and blue gain multipliers, 0 through 1023/256.
`lsc`	`enable`, `lsc_gain_size`, `gain_r`, `gain_gr`, `gain_gb`, `gain_b`: four gain lists, each with the length returned by `get_isp("lsc")`; values are 0 through 1023/256. The size is determined by capture dimensions.
`blc`	`enable`, `stretch_enable`, `top_left_offset`, `top_right_offset`, `bottom_left_offset`, `bottom_right_offset`: black level offsets, integers from 0 through 255.
`bf`	`enable`, `level` (2–20), `matrix`: nine integers from 0 through 15 in row order.
`sharpen`	`enable`, `h_thresh`, `l_thresh` (0–255, low ≤ high), `h_coeff`, `m_coeff` (0 through 255/32), `matrix`: nine integers from 0 through 31 in row order.
`ccm`	`enable`, `matrix`: nine finite numbers strictly between -4 and 4 in row order.
`gamma`	`enable`, `x`, `y`: two lists of 16 integers from 0 through 255. X must increase from an implicit zero by powers of two; the last point must be 255 and is treated as 256 for the final interval.

Set `enable=True` when applying new parameters to a disabled block. Passing only `enable=False` disables processing without replacing its stored tables. White balance gains are kept across that disable and come back when the block is enabled again. Gamma readback reports the red channel; setting `x` or `y` applies the resulting curve to all three channels, while an enable-only update preserves the separate channel curves. LSC tables are copied into camera-owned storage and remain valid across garbage collection and IPA switches; the camera keeps the previously installed table intact until the driver accepts the new one, but the underlying driver can update its cached pointers before reporting a hardware error. A failed device write can therefore leave configuration or hardware state uncertain; reset the camera before continuing after such an error. Fixed-point values and sensor settings may be quantized by the driver.

Unknown blocks, invalid values, and array length mismatches are rejected before applying the update. An unsupported write raises `OSError`. A device error may leave a partially applied configuration; use `sensor.reset()` to reinitialize after such a failure.

The `example/01-Camera/04-IPA/isp_preview.py` example configures the output format, disables IPA, then continuously previews while randomizing exposure, gain, white balance gains, and CCM every 2 seconds until Ctrl-C. Each update prints the requested exposure and the driver readback for gain, white balance gains, and CCM. Adjust `UPDATE_INTERVAL_MS` and the parameter range constants to change the experiment; a value the active sensor rejects is printed and skipped rather than stopping the preview. Ctrl-C leaves IPA disabled and preserves the last settings.

See also:

The Camera Pipeline explains how a frame travels from the image sensor to an `image.Image`, and *The Image Model* covers pixel formats and color spaces.

Runnable examples: `example/01-Camera/00-Snapshot` (capture and save) and `example/01-Camera/03-MJPEG` (Wi-Fi MJPEG stream).

6.2.7 Constants

`sensor.`**`GRAYSCALE`**

Pixel format constants accepted by set_pixformat().

`sensor.`**`RGB565`**

`sensor.`**`QQVGA`**

Frame size constants accepted by set_framesize().

`sensor.`**`QVGA`**

`sensor.`**`VGA`**

6.2.8 Functions

`sensor.`**`reset`**`()`

Reset and initialize the camera with the board default sensor configuration.

`sensor.`**`shutdown`**`(`*enable=...*`)`

Stop or restart the camera stream.

Parameters

`enable` –True shuts the camera down; False starts it again.

`sensor.`**`set_pixformat`**`(`*pixformat*`)`

Select the output pixel format.

Parameters

`pixformat` –sensor.GRAYSCALE or sensor.RGB565.

`sensor.`**`get_pixformat`**`()`

Return the current pixel format constant.

`sensor.`**`set_framesize`**`(`*framesize*`)`

Select the output frame size.

Parameters

`framesize` –sensor.QQVGA, sensor.QVGA, or sensor.VGA when supported by the board.

`sensor.`**`get_framesize`**`()`

Return the current frame size constant.

`sensor.`**`width`**`()`

Return the current output image width in pixels.

sensor.**height**()

Return the current output image height in pixels.

sensor.**get_id**()

Return the camera sensor ID.

sensor.**set_hmirror**(*enable*)

Mirror the camera image horizontally.

Parameters

enable –True enables horizontal mirror.

sensor.**get_hmirror**()

Return whether horizontal mirror is enabled.

sensor.**set_vflip**(*enable*)

Flip the camera image vertically.

Parameters

enable –True enables vertical flip.

sensor.**get_vflip**()

Return whether vertical flip is enabled.

sensor.**set_ipa**(*enable*)

Optional API: present only in firmware with IPA/ISP control enabled. Start or stop the IPA controller after camera initialization; briefly pauses capture. Existing ISP tuning is retained when disabled; enabling returns configured blocks to IPA. IPA controls only algorithms declared in its active JSON configuration and supported by the backend. Sensor-side automatic exposure/white balance is independent. Unsupported cameras raise OSError. OV2710 exposure/gain remain sensor-controlled when IPA is stopped; manual controls stay unsupported.

sensor.**get_ipa**()

Optional API: present only in firmware with IPA/ISP control enabled. Return the IPA controller state, or None when runtime control is unsupported. A supported camera must be initialized first.

sensor.**get_ipa_stats**(*timeout=...*)

Optional API: collect one new statistics packet while IPA is running. Uninitialized/stopped cameras or disabled IPA raise OSError. AE and AWB subwindow grids use [x][y]; unavailable blocks are None.

Parameters

timeout –1..1000 milliseconds; None means timeout or unsupported backend.

sensor.**get_isp**(*block*)

Optional API: present only in firmware with IPA/ISP control enabled. Read current ISP block parameters, or None if unsupported. Initialize the camera first. Blocks the camera driver or chip revision does not implement return None, including "exposure" on sensors whose driver makes it write-only. Readback is driver configuration, not a frame commit barrier or sensor-side AWB gains. OV2710 does not support manual exposure or pixel_gain reads or writes.

sensor.**set_isp**(*block*, ***params*)

Optional API: present only in firmware with IPA/ISP control enabled. Update one ISP block using keyword parameters; unspecified fields retain their values. Disable IPA before writing. Invalid parameters raise ValueError; device errors raise OSError. BF level is 2..20; use enable=False to disable filtering. Consume queued frames before evaluating a change; skip_frames is not a hardware commit barrier.

sensor.**skip_frames**(*time=...*, *n=...*)

Drop frames while camera exposure and processing settle.

Parameters

time –milliseconds to wait. n: number of frames to skip.

sensor.**snapshot**(*buffer=..., /*)

Capture one image from the camera.

Parameters

buffer –optional writable buffer of at least the configured frame size. The returned Image retains it.

sensor.**status**()

Return camera readiness, size, format, mirror, flip, and crop status.

6.2.9 Classes

class sensor.**SensorStatus**

Dictionary returned by status().

class sensor.**IPAAWBStats**

class sensor.**IPAAFStats**

class sensor.**IPAStats**

6.3 image –Image Processing

The image module provides the Image object and the vision algorithms built on OpenMV imlib: drawing, format conversion, filtering, color/blob analysis, and feature detection (lines, line segments, circles, rectangles, QR codes, and AprilTags). The codec stream type *imageio.ImageIO* is documented in *image.ImageIO –Image Stream*.

The current ESP32-P4 board profiles also enable the ZXing-C++ barcode backend and *image.Image.find_barcodes()*.

6.3.1 Drawing and Color-Blob Tracking

```
import sensor

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.skip_frames(time=1000)

img = sensor.snapshot()
img.draw_rectangle(10, 10, 50, 50, color=(255, 0, 0))
thresholds = [(30, 100, 15, 127, 15, 127)]
for blob in img.find_blobs(thresholds, pixels_threshold=80, area_threshold=80):
    img.draw_rectangle(blob.rect(), color=(255, 0, 0), thickness=2)
    img.draw_cross(blob.cx(), blob.cy(), color=(0, 255, 0))
```

An RGB565 threshold contains six LAB limits: (L_min, L_max, A_min, A_max, B_min, B_max). pixels_threshold rejects regions containing too few matching pixels, while area_threshold rejects regions whose bounding box is too small. Blob coordinates are returned in source-image pixels and can be passed directly to drawing methods.

6.3.2 Filtering and Binary Segmentation

```
sensor.set_pixformat(sensor.GRAYSCALE)
img = sensor.snapshot()
img.gaussian(1)
img.binary([(80, 255)])
img.erode(1)
img.dilate(1)
img.flush()
```

`gaussian()` suppresses high-frequency noise before thresholding. `binary()` converts matching grayscale values to the active binary value, and the erosion/dilation pair removes isolated pixels and restores the main regions. These methods modify the image in place.

6.3.3 QR Code Recognition

```
sensor.set_pixformat(sensor.GRAYSCALE)
img = sensor.snapshot()
for code in img.find_qrcodes():
    img.draw_rectangle(code.rect(), color=255, thickness=2)
    print(code.payload())
```

`find_qrcodes()` returns geometry and decoded payload data. Grayscale input generally reduces processing cost, and a smaller ROI can be supplied when the code is expected in a known part of the frame.

6.3.4 Line, Segment, and Circle Detection

```
img = sensor.snapshot()
for line in img.find_lines(threshold=1400, theta_margin=25, rho_margin=25):
    img.draw_line(line.line(), color=255, thickness=2)

for segment in img.find_line_segments(threshold=50, merge_distance=10):
    img.draw_line(segment.line(), color=255, thickness=2)

for circle in img.find_circles(threshold=2500, r_min=8, r_max=80, r_step=4):
    img.draw_circle(circle.x(), circle.y(), circle.r(), color=255, thickness=2)
```

`find_lines()` and `find_circles()` use Hough accumulator thresholds. `find_line_segments()` uses the Edge Drawing Lines detector, where `threshold` is the minimum Sobel gradient magnitude and `merge_distance` joins nearby segments. Increasing thresholds reduces weak detections; constraining the radius range, ROI, or image resolution lowers computation and usually improves stability.

6.3.5 Encode and Save an Image

```
import image

img = sensor.snapshot()
jpg = img.to_jpeg(quality=85, subsampling=image.JPEG_SUBSAMPLING_420)
jpg.save("/sdcard/snapshot.jpg")
print("encoded bytes:", jpg.size())
```

JPEG quality trades encoded size for detail, and 4:2:0 chroma subsampling usually produces the smallest color image. Saving requires the destination filesystem to be mounted and writable.

Most in-place methods return the image itself, so operations can be chained: `img.to_grayscale().gaussian(1).binary([(128, 255)])`.

See also:

For the theory behind these methods, see *The Image Model* (pixel formats, color spaces, the frame buffer) and *Image Processing* (filtering, thresholding, feature detection).

Runnable examples: `example/02-Image-Processing` (drawing, filters, color tracking, frame differencing), `example/04-Barcodes` (QR codes), and `example/05-Feature-Detection` (AprilTags, lines, line segments, circles).

6.3.6 Constants

`image.`**`BINARY`**
: 1-bpp black-and-white pixel format (one bit per pixel).

`image.`**`GRAYSCALE`**
: 8-bit grayscale pixel format.

`image.`**`RGB565`**
: 16-bit RGB565 color pixel format.

`image.`**`BAYER`**
: Raw Bayer mosaic pixel format from the sensor.

`image.`**`YUV422`**
: Packed YUV422 pixel format.

`image.`**`JPEG`**
: JPEG-compressed image format.

`image.`**`PNG`**
: PNG-compressed image format.

`image.`**`PALETTE_RAINBOW`**
: False-color rainbow palette (cold to hot).

`image.`**`PALETTE_IRONBOW`**
: "Ironbow" thermal palette.

`image.`**`PALETTE_DEPTH`**
: Depth-map palette.

`image.`**`PALETTE_EVT_DARK`**
: Event-camera dark palette.

`image.`**`PALETTE_EVT_LIGHT`**
: Event-camera light palette.

`image.`**`AREA`**
: Scaling/geometry hint flags for copy()/crop()/scale()/draw_image(): area-average downscaling.

`image.`**`BILINEAR`**
: Bilinear interpolation hint.

`image.`**`BICUBIC`**
: Bicubic interpolation hint.

`image.`**`HMIRROR`**
: Mirror horizontally.

`image.`**`VFLIP`**
: Flip vertically.

`image.`**`TRANSPOSE`**
: Swap X and Y (transpose).

`image.`**`CENTER`**
: Center the source in the destination.

`image.`**`EXTRACT_RGB_CHANNEL_FIRST`**
: Extract the RGB channel before applying the color palette.

`image.`**`APPLY_COLOR_PALETTE_FIRST`**
: Apply the color palette before other steps.

`image.`**`SCALE_ASPECT_KEEP`**
: Keep aspect ratio, fitting inside the destination.

`image.`**`SCALE_ASPECT_EXPAND`**
: Keep aspect ratio, expanding to fill the destination.

`image.`**`SCALE_ASPECT_IGNORE`**
: Ignore aspect ratio, stretching to the destination.

`image.`**`BLACK_BACKGROUND`**
: Treat the background as black (needed for correct alpha blending).

`image.`**`ROTATE_90`**
: Rotate 90 degrees.

`image.`**`ROTATE_180`**
: Rotate 180 degrees.

`image.`**`ROTATE_270`**
: Rotate 270 degrees.

`image.`**`JPEG_SUBSAMPLING_AUTO`**
: Let the JPEG encoder pick chroma subsampling automatically.

`image.`**`JPEG_SUBSAMPLING_444`**
: 4:4:4 JPEG chroma subsampling (best quality).

`image.`**`JPEG_SUBSAMPLING_422`**
: 4:2:2 JPEG chroma subsampling.

`image.`**`JPEG_SUBSAMPLING_420`**
: 4:2:0 JPEG chroma subsampling (smallest size).

`image.`**`SEARCH_EX`**
: Exhaustive template-search strategy.

`image.`**`SEARCH_DS`**
: Diamond-search (faster, approximate) template-search strategy.

`image.`**`EDGE_CANNY`**
: Canny edge-detection method for find_edges().

`image.`**`EDGE_SIMPLE`**
: Simple gradient-threshold edge-detection method for find_edges().

`image.`**`CORNER_FAST`**
: FAST corner detector for find_keypoints().

`image.`**`CORNER_AGAST`**
: AGAST corner detector for find_keypoints().

image.**EAN2**
Barcode symbology constants returned by barcode.type() / accepted by find_barcodes().

image.**EAN5**

image.**EAN8**

image.**UPCE**

image.**ISBN10**

image.**UPCA**

image.**EAN13**

image.**ISBN13**

image.**I25**

image.**DATABAR**

image.**DATABAR_EXP**

image.**CODABAR**

image.**CODE39**

image.**PDF417**

image.**CODE93**

image.**CODE128**

image.**TAG16H5**
AprilTag family constants accepted by find_apriltags().

image.**TAG25H7**

image.**TAG25H9**

image.**TAG36H10**

image.**TAG36H11**

6.3.7 Functions

image.**binary_to_grayscale**(*pixel*)
Convert a 1-bpp binary pixel to a grayscale value.

image.**binary_to_rgb**(*pixel*)
Convert a 1-bpp binary pixel to an (r, g, b) tuple.

image.**binary_to_lab**(*pixel*)
Convert a 1-bpp binary pixel to a LAB tuple.

image.**binary_to_yuv**(*pixel*)
Convert a 1-bpp binary pixel to a YUV tuple.

image.**grayscale_to_binary**(*pixel*)
Convert a grayscale value to a 1-bpp binary pixel.

image.**grayscale_to_rgb**(*pixel*)
Convert a grayscale value to an (r, g, b) tuple.

image.**grayscale_to_lab**(*pixel*)
Convert a grayscale value to a LAB tuple.

image.**grayscale_to_yuv**(*pixel*)
Convert a grayscale value to a YUV tuple.

image.**rgb_to_binary**(*pixel*)
Convert an (r, g, b) color to a 1-bpp binary pixel.

image.**rgb_to_grayscale**(*pixel*)
Convert an (r, g, b) color to a grayscale value.

image.**rgb_to_lab**(*pixel*)
Convert an (r, g, b) color to a LAB tuple.

image.**rgb_to_yuv**(*pixel*)
Convert an (r, g, b) color to a YUV tuple.

image.**lab_to_binary**(*pixel*)
Convert a LAB tuple to a 1-bpp binary pixel.

image.**lab_to_grayscale**(*pixel*)
Convert a LAB tuple to a grayscale value.

image.**lab_to_rgb**(*pixel*)
Convert a LAB tuple to an (r, g, b) tuple.

image.**lab_to_yuv**(*pixel*)
Convert a LAB tuple to a YUV tuple.

image.**yuv_to_binary**(*pixel*)
Convert a YUV tuple to a 1-bpp binary pixel.

image.**yuv_to_grayscale**(*pixel*)
Convert a YUV tuple to a grayscale value.

image.**yuv_to_rgb**(*pixel*)
Convert a YUV tuple to an (r, g, b) tuple.

image.**yuv_to_lab**(*pixel*)
Convert a YUV tuple to a LAB tuple.

6.3.8 Classes

class image.**line**
A detected line segment with Hough-space attributes.

line()
Return the line as (x1, y1, x2, y2).

x1()
Start-point x coordinate.

y1()
Start-point y coordinate.

x2()
End-point x coordinate.

y2()
End-point y coordinate.

length()
Segment length in pixels.

magnitude()
Hough accumulator magnitude (line strength).

theta()
Line angle theta in degrees (Hough space).

rho()
Line distance rho in pixels (Hough space).

class image.**circle**
A detected circle.

circle()
Return the circle as (x, y, r).

x()
Center x coordinate.

y()
Center y coordinate.

r()
Radius in pixels.

magnitude()
Hough accumulator magnitude (circle strength).

class image.**rect**
A detected rectangle (quadrilateral).

corners()
Return the four corner points in clockwise order.

rect()
Return the bounding box as (x, y, w, h).

x()
Bounding-box top-left x.

y()
Bounding-box top-left y.

w()
Bounding-box width.

h()
Bounding-box height.

magnitude()
Detection strength.

class image.**blob**
A connected color region found by find_blobs().

corners()
The four bounding-box corners.

min_corners()
The four corners of the minimum-area enclosing rectangle.

rect()
Bounding box as (x, y, w, h).

x()
Bounding-box top-left x.

y()
Bounding-box top-left y.

w()
Bounding-box width.

h()
Bounding-box height.

pixels()
Number of pixels in the blob.

cx()
Centroid x (integer).

cxf()
Centroid x (float).

cy()
Centroid y (integer).

cyf()
Centroid y (float).

rotation()
Orientation of the blob' s major axis in radians.

rotation_deg()
Orientation in degrees.

rotation_rad()
Orientation in radians.

code()
Bitmask of the thresholds this blob matched.

count()
Number of blobs merged into this one.

perimeter()
Bounding-box perimeter.

roundness()
Roundness metric in [0, 1] (1 == circle).

elongation()
Elongation metric in [0, 1].

area()
Bounding-box area (w * h).

density()
Pixel density: pixels / area.

extent()
Alias of density().

compactness()
Compactness metric in [0, 1].

solidity()
Solidity (pixels / convex-hull area) in [0, 1].

convexity()
Convexity metric in [0, 1].

x_hist_bins()
Horizontal projection histogram bins.

y_hist_bins()
Vertical projection histogram bins.

major_axis_line()
Major-axis line as (x1, y1, x2, y2).

minor_axis_line()
Minor-axis line as (x1, y1, x2, y2).

enclosing_circle()
Minimum enclosing circle as (x, y, r).

enclosed_ellipse()
Best-fit enclosed ellipse as (x, y, rx, ry, rotation).

class image.**qrcode**
A decoded QR code found by find_qrcodes().

corners()
The four corner points.

rect()
Bounding box as (x, y, w, h).

x()
Bounding-box top-left x.

y()
Bounding-box top-left y.

w()
Bounding-box width.

h()
Bounding-box height.

payload()
Decoded payload string.

version()
QR version (size) number.

ecc_level()
Error-correction level.

mask()
Data mask pattern index.

data_type()
Encoding mode of the payload.

eci()

Extended Channel Interpretation value.

is_numeric()

True if the payload is numeric.

is_alphanumeric()

True if the payload is alphanumeric.

is_binary()

True if the payload is binary.

is_kanji()

True if the payload is Kanji.

class image.**barcode**

A decoded 1D/2D barcode found by find_barcodes().

corners()

The four corner points.

rect()

Bounding box as (x, y, w, h).

x()

Bounding-box top-left x.

y()

Bounding-box top-left y.

w()

Bounding-box width.

h()

Bounding-box height.

payload()

Decoded payload string.

type()

Symbology type constant (e.g. EAN13, CODE128).

rotation()

Rotation of the barcode in degrees.

quality()

Decode quality (number of agreeing scan lines).

class image.**apriltag**

A detected AprilTag with optional 6-DoF pose. Attributes are fields, not methods.

class image.**histogram**

Channel histogram returned by get_histogram(). For RGB565 the L/A/B channels correspond to the LAB color space.

bins()

Combined bins (grayscale/binary) or L bins (RGB565). Values are normalized floating-point pixel frequencies, not integer counts. For a non-empty selection their sum is approximately 1.0 (floating-point rounding applies).

l_bins()

L-channel (lightness) bins.

a_bins()

A-channel bins (RGB565 only).

b_bins()

B-channel bins (RGB565 only).

get_percentile(*p*)

Return the value at the given cumulative percentile (0..1).

Parameters

p – percentile in the range 0 to 1.

get_threshold()

Compute an Otsu threshold from the histogram.

get_stats()

Compute statistics (mean/median/mode/stdev/…) from the histogram.

get_statistics()

Alias of get_stats().

statistics()

Alias of get_stats().

class image.**percentile**

A percentile result from histogram.get_percentile().

value()

Combined/L value at the percentile.

l_value()

L-channel value.

a_value()

A-channel value.

b_value()

B-channel value.

class image.**threshold**

An Otsu threshold result from histogram.get_threshold().

value()

Combined/L threshold value.

l_value()

L-channel threshold.

a_value()

A-channel threshold.

b_value()

B-channel threshold.

class image.**statistics**

Region statistics returned by get_statistics(). Combined accessors apply to grayscale/binary; `l_*`/`a_*`/`b_*` apply to RGB565 LAB channels.

mean()

median()

mode()

stdev()

min()

max()

lq()
Lower quartile (25th percentile).

uq()
Upper quartile (75th percentile).

l_mean()

l_median()

l_mode()

l_stdev()

l_min()

l_max()

l_lq()

l_uq()

a_mean()

a_median()

a_mode()

a_stdev()

a_min()

a_max()

a_lq()

a_uq()

b_mean()

b_median()

b_mode()

b_stdev()

b_min()

b_max()

b_lq()

b_uq()

class image.**Image**(*path*)

class image.**Image**(*width*, *height*, *pixformat*, *, *buffer*=...)

class image.**Image**(*array*, *, *buffer=...*)

A 2D image backed by a pixel buffer. Create one from a file path, from a width/height/pixformat, or from an array; most often you get one from sensor.snapshot(). Most in-place methods return `self` so calls can be chained.

Load an image from a file (BMP/PPM/PGM/JPEG/PNG depending on build).

Parameters

path –source file path.

width()

Image width in pixels.

height()

Image height in pixels.

format()

Pixel format constant (BINARY/GRAYSCALE/RGB565/⋯).

size()

Size of the pixel buffer in bytes.

bytearray()

Return the pixel buffer as a bytearray (no copy).

get_pixel(*x*, *y*, *, *rgbtuple=...*)

Read one pixel. x, y: pixel coordinates.

Parameters

rgbtuple –for RGB565, True returns an (r, g, b) tuple, False the packed value.

set_pixel(*x*, *y*, *pixel*)

Write one pixel. x, y: pixel coordinates.

Parameters

pixel –grayscale value or (r, g, b) color.

flush()

Push this image to the host as a binary EV-MUX `preview.frame` JPEG.

save(*path*, *, *roi=...*, *quality=...*)

Save the image to a file; the format is inferred from the extension. PNG encoding is not enabled in current firmware.

Parameters

- **path** –destination path.
- **roi** –optional source rectangle (x, y, w, h).
- **quality** –JPEG quality from 1 to 100 when saving as JPEG.

copy(*, *roi=...*, *x_scale=...*, *y_scale=...*, *rgb_channel=...*, *alpha=...*, *color_palette=...*, *alpha_palette=...*, *hint=...*, *copy=...*)

Return a scaled/cropped/converted copy of the image. x_scale, y_scale: horizontal/vertical scale factors.

Parameters

- **roi** –source rectangle (x, y, w, h) to read from.
- **hint** –bitwise OR of AREA/BILINEAR/BICUBIC/HMIRROR/VFLIP/TRANSPOSE/ROTATE_*/SCALE_ flags.
- **rgb_channel** –extract a single RGB channel (0, 1, or 2; -1 for all).
- **alpha** –blend alpha from 0 to 255.
- **copy** –True returns a new image; False converts in place.

crop(*, *roi=...*, *x_scale=...*, *y_scale=...*, *hint=...*, *copy=...*)

Crop/scale the image to a region of interest. Same options as copy().

scale(*, *x_scale=...*, *y_scale=...*, *roi=...*, *hint=...*, *copy=...*)

Scale the image by x_scale/y_scale (alias of crop in this build).

compress(*, *quality=...*, *subsampling=...*, *roi=...*)

Compress the image to JPEG in place (alias of to_jpeg).

Parameters

- **quality** –JPEG quality from 1 to 100.
- **subsampling** –one of the JPEG_SUBSAMPLING_* constants.

to_bitmap(*, *copy=...*, *roi=...*)

Convert to a 1-bpp bitmap.

to_grayscale(*, *copy=...*, *roi=...*)

Convert to grayscale.

to_rgb565(*, *copy=...*, *roi=...*)

Convert to RGB565.

to_rainbow(*, *copy=...*, *roi=...*)

Apply the rainbow palette and convert to RGB565.

to_ironbow(*, *copy=...*, *roi=...*)

Apply the ironbow thermal palette and convert to RGB565.

to_jpeg(*, *quality=...*, *subsampling=...*, *copy=...*)

Encode to JPEG.

Parameters

- **quality** –JPEG quality from 1 to 100.
- **subsampling** –one of the JPEG_SUBSAMPLING_* constants.

to_png(*, *copy=...*)

Encode to PNG (not enabled in current firmware).

clear(*, *mask=...*)

Fill the image with zeros (optionally only where mask is set).

Parameters

mask –optional 1-bpp image limiting which pixels are cleared.

draw_line(*x0*, *y0*, *x1*, *y1*, *color=...*, *thickness=...*)

draw_line(*line*, *color=...*, *thickness=...*)

draw_rectangle(*x*, *y*, *w*, *h*, *color=...*, *thickness=...*, *fill=...*)

draw_rectangle(*rect*, *color=...*, *thickness=...*, *fill=...*)

draw_circle(*x*, *y*, *r*, *color=...*, *thickness=...*, *fill=...*)

draw_circle(*circle*, *color=...*, *thickness=...*, *fill=...*)

draw_ellipse(*x*, *y*, *rx*, *ry*, *rotation*, *color=...*, *thickness=...*, *fill=...*)

draw_ellipse(*ellipse*, *color=...*, *thickness=...*, *fill=...*)

draw_string(*x*, *y*, *text*, *color=...*, *scale=...*, *x_spacing=...*, *y_spacing=...*, *mono_space=...*, *char_rotation=...*, *char_hmirror=...*, *char_vflip=...*, *string_rotation=...*, *string_hmirror=...*, *string_vflip=...*)

Draw text using the built-in bitmap font. x, y: top-left position of the text.

Parameters

- **text** –string to draw.
- **color** –text color.
- **scale** –integer/float scale factor of the font.

draw_cross(*x*, *y*, *color=...*, *size=...*, *thickness=...*)

Draw a cross marker centered at (x, y).

draw_arrow(*x0*, *y0*, *x1*, *y1*, *color=...*, *size=...*, *thickness=...*)

draw_arrow(*line*, *color=...*, *size=...*, *thickness=...*)

draw_edges(*corners*, *color=...*, *size=...*, *thickness=...*, *fill=...*)

Draw connecting edges between a sequence of corner points.

draw_keypoints(*keypoints*, *color=...*, *size=...*, *thickness=...*, *fill=...*)

Draw a set of keypoints.

draw_image(*image*, *x=...*, *y=...*, ***, *x_scale=...*, *y_scale=...*, *roi=...*, *rgb_channel=...*, *alpha=...*, *color_palette=...*, *alpha_palette=...*, *hint=...*, *mask=...*)

Alpha-blend/scale another image (or a solid color) onto this one. x, y: destination top-left position. x_scale, y_scale: scale factors; roi: source rectangle; alpha: 0..255;

Parameters

- **image** –source Image, an (r, g, b) tuple, or a packed scalar color.
- **hint** –SCALE_ASPECT_*/interpolation flags; mask: optional 1-bpp mask.

binary(*thresholds*, ***, *invert=...*, *zero=...*, *mask=...*, *to_bitmap=...*, *copy=...*)

Threshold the image into a binary mask against a list of color thresholds.

Parameters

- **thresholds** –list of grayscale (min, max) or LAB 6-tuple thresholds.
- **invert** –invert the match.
- **zero** –zero out matching pixels instead of setting them.
- **mask** –optional 1-bpp image limiting the operation.
- **to_bitmap** –output a 1-bpp bitmap.
- **copy** –return a new image instead of modifying in place.

invert()

Invert pixel values.

erode(*ksize*, ***, *threshold=...*, *mask=...*)

Morphological erosion with a (2*ksize+1) square structuring element.

Parameters

- **ksize** –structuring-element radius.
- **threshold** –minimum number of set neighbors to keep a pixel.
- **mask** –optional 1-bpp image limiting the operation.

dilate(*ksize*, ***, *threshold=...*, *mask=...*)

Morphological dilation. See erode() for parameters.

open(*ksize*, ***, *threshold=...*, *mask=...*)

Morphological opening (erode then dilate).

close(*ksize*, ***, *threshold=...*, *mask=...*)

Morphological closing (dilate then erode).

difference(*image*, *x=...*, *y=...*, ***, *roi=...*, *mask=...*)

Absolute per-pixel difference against another image/color (frame differencing). x, y: placement of the other image.

Parameters

image –other Image, (r, g, b) tuple, or scalar color.

histeq(***, *adaptive=...*, *clip_limit=...*, *mask=...*)

Histogram equalization (global). adaptive is not supported in this build.

Parameters

clip_limit –reserved; mask: optional 1-bpp image.

mean (*ksize*, *, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Box (mean) blur over a (2*ksize+1) window. threshold/offset/invert: adaptive-threshold the result.

Parameters

- **ksize** –kernel radius.
- **mask** –optional 1-bpp image.

median (*ksize*, *, *percentile=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Median filter (edge-preserving denoise).

Parameters

- **ksize** –kernel radius.
- **percentile** –rank in [0, 1] to select (0.5 == median).

mode (*ksize*, *, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Mode (most-common value) filter.

Parameters

ksize –kernel radius.

midpoint (*ksize*, *, *bias=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Midpoint filter ((min + max)/2 blended by bias).

Parameters

ksize –kernel radius; bias: 0 == min, 1 == max, 0.5 == midpoint.

morph (*ksize*, *kernel*, *, *mul=...*, *add=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Apply an arbitrary NxN convolution kernel.

Parameters

- **ksize** –kernel radius; the kernel must be (2*ksize+1) square.
- **kernel** –flat or nested sequence of integer weights.
- **mul** –output multiplier (defaults to 1/sum(kernel)); add: output bias.

blur (*ksize*, *, *unsharp=...*, *mul=...*, *add=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Gaussian blur using a separable Pascal-triangle kernel (alias: gaussian, gaussian_blur).

Parameters

- **ksize** –kernel radius.
- **unsharp** –True turns the filter into an unsharp-mask sharpener.

gaussian (*ksize*, *, *unsharp=...*, *mul=...*, *add=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Gaussian blur. See blur() for parameters.

gaussian_blur (*ksize*, *, *unsharp=...*, *mul=...*, *add=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Gaussian blur. See blur() for parameters.

laplacian (*ksize*, *, *sharpen=...*, *mul=...*, *add=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Laplacian edge filter (or sharpener when sharpen is True).

Parameters

ksize –kernel radius.

bilateral (*ksize*, *, *color_sigma=...*, *space_sigma=...*, *threshold=...*, *offset=...*, *invert=...*, *mask=...*)

Bilateral filter (edge-preserving smoothing).

Parameters

- **ksize** –kernel radius.
- **color_sigma** –range sigma (color closeness); larger blurs across more contrast.
- **space_sigma** –spatial sigma (distance closeness).

get_histogram (*thresholds=...*, *, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Compute a channel histogram over the image or a region. bins / l_bins / a_bins / b_bins: bin counts; -1 selects the channel default.

Parameters

- **thresholds** –optional list of thresholds to restrict counted pixels.
- **invert** –invert the threshold match.
- **roi** –region of interest (x, y, w, h).
- **difference** –optional image to histogram the difference against.

get_hist(*thresholds=...*, ***, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Alias of get_histogram().

histogram(*thresholds=...*, ***, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Alias of get_histogram().

get_statistics(*thresholds=...*, ***, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Compute region statistics (mean/median/mode/stdev/quartiles). thresholds/invert/roi/bins: as in get_histogram().

get_stats(*thresholds=...*, ***, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Alias of get_statistics().

statistics(*thresholds=...*, ***, *invert=...*, *roi=...*, *bins=...*, *l_bins=...*, *a_bins=...*, *b_bins=...*, *difference=...*)

Alias of get_statistics().

get_regression(*thresholds*, ***, *invert=...*, *roi=...*, *x_stride=...*, *y_stride=...*, *area_threshold=...*, *pixels_threshold=...*, *robust=...*)

Fit a line to the thresholded pixels via least-squares (or robust) regression. x_stride, y_stride: pixel sampling steps. area_threshold, pixels_threshold: minimum region area / pixel count to fit.

Parameters

- **thresholds** –list of color thresholds selecting the pixels to fit.
- **invert** –invert the match; roi: region of interest.
- **robust** –use a robust (Theil-Sen) estimator.

find_blobs(*thresholds*, ***, *invert=...*, *roi=...*, *x_stride=...*, *y_stride=...*, *area_threshold=...*, *pixels_threshold=...*, *merge=...*, *margin=...*, *threshold_cb=...*, *merge_cb=...*, *x_hist_bins_max=...*, *y_hist_bins_max=...*)

Find connected color regions (blobs) matching the thresholds. x_stride, y_stride: pixel sampling steps. area_threshold, pixels_threshold: minimum bounding-box area / pixel count. threshold_cb / merge_cb: optional Python filter/merge callbacks.

Parameters

- **thresholds** –list of color thresholds.
- **invert** –invert the match; roi: region of interest.
- **merge** –merge overlapping blobs; margin: extra merge margin.

find_lines(***, *roi=...*, *x_stride=...*, *y_stride=...*, *threshold=...*, *theta_margin=...*, *rho_margin=...*)

Find straight lines with the Hough transform. theta_margin, rho_margin: merge tolerance for similar lines.

Parameters

- **roi** –region of interest; x_stride, y_stride: sampling steps.
- **threshold** –minimum Hough accumulator magnitude to report a line.

find_line_segments(***, *roi=...*, *merge_distance=...*, *max_theta_diff=...*, *threshold=...*)

Find line segments with the Edge Drawing Lines detector. merge_distance and max_theta_diff merge nearby, similarly angled segments.

Parameters

threshold –minimum Sobel gradient magnitude used to seed and trace edges.

find_circles(***, *roi=...*, *x_stride=...*, *y_stride=...*, *threshold=...*, *x_margin=...*, *y_margin=...*, *r_margin=...*, *r_min=...*, *r_max=...*, *r_step=...*)

Find circles with the Hough transform. x_margin, y_margin, r_margin: merge tolerances. r_min, r_max, r_step: radius search range and step.

Parameters
threshold –minimum accumulator magnitude.

find_rects(*, *roi*=..., *threshold*=...)

Find rectangles/quadrilaterals (e.g. for fiducials).

Parameters
threshold –minimum edge-magnitude score.

find_qrcodes(*, *roi*=...)

Find and decode QR codes.

Parameters
roi –region of interest.

find_barcodes(*, *roi*=...)

Find and decode 1D/2D barcodes (requires the barcode backend).

Parameters
roi –region of interest.

find_apriltags(*, *roi*=..., *families*=..., *fx*=..., *fy*=..., *cx*=..., *cy*=..., *pose*=...)

Find and decode AprilTags, optionally estimating 6-DoF pose. fx, fy, cx, cy: camera intrinsics for pose estimation.

Parameters
- **roi** –region of interest; families: OR of TAG* family constants.
- **pose** –True computes translation/rotation (needs intrinsics).

6.4 display –LCD Output

The `display` module drives the board LCD. Construct one `ESP32Display` (aliased as `display.Display`) and push frames to it with `ESP32Display.write()`.

6.4.1 Camera Preview

```
import sensor, display

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)

lcd = display.ESP32Display()
while True:
    lcd.write(sensor.snapshot(), fit=True)
```

With `fit=True`, the source image is scaled to fit the panel while preserving the display driver's default placement behavior. Create the display once and reuse it; repeatedly constructing the object reinitializes panel resources.

6.4.2 Drawing into a Reusable Image

For application-generated graphics, allocate a regular RGB565 image buffer once, update it, and pass it to `lcd.write()`:

```
import display, image

lcd = display.Display()
canvas = image.Image(lcd.width(), lcd.height(), image.RGB565)
canvas.draw_rectangle(0, 0, canvas.width(), canvas.height(), color=(6, 8, 18),␣
↪fill=True)
canvas.draw_string(20, 20, "ESP-VISION", color=(255, 255, 255), scale=2)
lcd.write(canvas, fit=False)
```

Keep and redraw the same `Image` when producing later frames. The display driver copies its pixels to the panel; the image does not need to be allocated in the driver's framebuffer.

6.4.3 Position, Scale, and Crop

Use `x` and `y` for placement, `x_scale` and `y_scale` for explicit scaling, and `roi` to display only a source region:

```
img = sensor.snapshot()
lcd.write(
    img,
    x=20,
    y=10,
    x_scale=0.5,
    y_scale=0.5,
    roi=(40, 30, 160, 120),
    fit=False,
)
```

When `fit=False`, the explicit position and scale values control the output. The ROI is expressed in source-image coordinates as `(x, y, width, height)`; reducing the ROI also reduces the amount of image data sent to the display pipeline.

6.4.4 Backlight and Cleanup

```
lcd = display.Display(backlight=80)
print("panel:", lcd.width(), "x", lcd.height())
lcd.backlight(30)
lcd.clear()
lcd.deinit()
```

Backlight values are percentages from 0 to 100. Call `deinit()` when an application permanently releases the display; do not write frames after the driver has been deinitialized.

See also:

Runnable example: `example/06-Peripherals/01-Display/lcd_preview.py`.

6.4.5 Classes

class display.**ESP32Display**(*width=...*, *height=...*, *refresh=...*, ***, *backlight=...*)

ESP32 display object for the board LCD.

Create and initialize the display.

Parameters

- **width** –requested display width; 0 uses board default.
- **height** –requested display height; 0 uses board default.
- **refresh** –target refresh rate in Hz.
- **backlight** –initial backlight percentage, from 0 to 100.

deinit()

Release the display driver resources.

width()

Return the physical display width in pixels.

height()

Return the physical display height in pixels.

clear(*display_off=...*)

Clear the display.

Parameters

display_off –True may turn the panel output off when supported by the board.

backlight(*value=...*)

Get or set backlight brightness.

Parameters

value –None returns current brightness; otherwise set 0 to 100 percent.

write(*image*, *, *x=...*, *y=...*, *x_scale=...*, *y_scale=...*, *roi=...*, *fit=...*)

Draw an image on the display. x, y: destination top-left position in display pixels. x_scale, y_scale: optional manual scale factors.

Parameters

- **image** –source image, normally from sensor.snapshot().
- **roi** –optional source rectangle (x, y, w, h).
- **fit** –True scales the image to fit the display area.

6.5 lcd_touch –Touch Input

The `lcd_touch` module reads the board's touchscreen independently of display initialization. `Touch()` requires configured touch hardware and raises `OSError` on unsupported boards.

6.5.1 Read Touch Coordinates

```
import lcd_touch
import time

touch = lcd_touch.Touch()
try:
    while True:
        points = touch.read()
        if points:
            print(points)
        time.sleep_ms(20)
finally:
    touch.deinit()
```

`read()` returns a tuple of `(x, y)` pixel coordinates, or `()` when there is no touch. Coordinates follow the board's touch configuration; image scaling, cropping, and placement must be accounted for by the application.

6.5.2 Ownership and Cleanup

The touch device is shared. Creating a new `Touch()` transfers ownership to that instance; older instances can no longer read it, and their `deinit()` calls do not release the new owner's device. Call `deinit()` when finished. Repeated cleanup is safe, and MicroPython soft reset also releases the device.

See also:

Runnable examples: `example/06-Peripherals/03-Touch/touch_read.py` and `example/06-Peripherals/03-Touch/touch_lcd.py`.

6.5.3 Classes

class lcd_touch.**Touch**

Touchscreen object for the board LCD. Requires a board with configured touch hardware; otherwise raises OSError.

Initialize touch input without requiring display initialization. A new instance takes ownership of the shared touch device.

read()

Return (x, y) pixel coordinates for each touch, or an empty tuple. Raise OSError if this instance is inactive or reading fails.

deinit()

Release the touch device if this instance still owns it.

6.6 espdl –Model Inference

The `espdl` module runs ESP-DL `.espdl` models on captured images. It provides task-specific wrappers for object detection (`PedestrianDetect`, `ESPDet`, `YOLO11`), pose estimation (`YOLO11nPose`), image classification (`ImageNetCls`), and single-object tracking (`NanoTrack`), plus `Model` for exposing raw ESP-DL output tensors to Python.

6.6.1 Button-Confirmed Single-Object Tracking

`espdl.NanoTrack` runs NanoTrack V2 full-INT8 models. Copy the three `.espdl` files from `models/nanotrack_v2/` to `/sdcard/nanotrack_v2/`. Local quantization experiments live under ignored `build/experiments/nanotrack-v2` and are not included in a clean checkout. Use the ESP32-P4 exports for P4/S31; the mixed INT16 experiment exports are not accepted by this wrapper. S3 requires target-specific quantization and separate validation.

`example/03-Machine-Learning/00-ESP-DL/nanotrack.py` displays a centered 64x64 selection box. Place the target inside it and press BOOT (GPIO35 on ESP32-P4X-EYE and P4 Function EV Board) to start tracking; press again to return to selection. Missed matches keep the original template and continue local searching. The camera image is displayed at native size, center-cropped on smaller LCDs. Change `BUTTON_PIN` and `BOX_SIZE` for other hardware or target sizes. Inference precedes drawing so overlays do not enter model inputs.

The example uses a configurable `MATCH_THRESHOLD = 0.95`. The displayed `match` value is a model score, not a target-identity probability.

```
from machine import Pin
import display
import espdl
import sensor
import time
```

(continues on next page)

(continued from previous page)

```
MODEL_DIR = "/sdcard/nanotrack_v2"
# Active-low BOOT button on ESP32-P4X-EYE and P4 Function EV Board.
# Change this pin for other boards.
BUTTON_PIN = 35
BOX_SIZE = 64
DEBOUNCE_MS = 30
# A match score is not a calibrated target-identity probability.
MATCH_THRESHOLD = 0.95


lcd = None
tracker = None
try:
    button = Pin(BUTTON_PIN, Pin.IN, Pin.PULL_UP)
    sensor.reset()
    sensor.set_pixformat(sensor.RGB565)
    sensor.set_framesize(sensor.QVGA)
    sensor.skip_frames(time=1000)
    lcd = display.Display()
    tracker = espdl.NanoTrack(
        MODEL_DIR + "/template_deploy.espdl",
        MODEL_DIR + "/search_deploy.espdl",
        MODEL_DIR + "/head_deploy.espdl",
    )
    frame = sensor.snapshot()
    width, height = frame.width(), frame.height()
    canvas = frame.copy()
    # Center at native size; smaller displays show the center crop.
    visible_w = min(width, lcd.width())
    visible_h = min(height, lcd.height())
    view = ((width - visible_w) // 2, (height - visible_h) // 2,
            visible_w, visible_h)
    offset_x = (lcd.width() - visible_w) // 2
    offset_y = (lcd.height() - visible_h) // 2
    size = min(BOX_SIZE, visible_w, visible_h)
    roi = ((width - size) // 2, (height - size) // 2, size, size)
    tracking = False
    last_button = stable_button = button.value()
    changed_at = time.ticks_ms()
    # A button held at startup must be released before it can confirm.
    armed = stable_button == 1

    while True:
        now = time.ticks_ms()
        level = button.value()
        pressed = False
        if level != last_button:
            last_button = level
            changed_at = now
        if level != stable_button and time.ticks_diff(now, changed_at) >= DEBOUNCE_
↪MS:
            stable_button = level
            if level:
                armed = True
            elif armed:
                armed = False
                pressed = True

        if pressed:
```

(continues on next page)

(continued from previous page)

```
            if tracking:
                tracking = False
            else:
                canvas.draw_image(frame, 0, 0)
                canvas.draw_string(view[0] + 2, view[1] + 2, "Initializing...",
                                   color=(255, 255, 0))
                lcd.write(canvas, x=offset_x, y=offset_y, roi=view, fit=False)
                # Use the last displayed raw frame, without any annotations.
                tracker.init(frame, roi)
                tracking = True

        frame = sensor.snapshot()
        begin = time.ticks_ms()
        result = tracker.track(frame, threshold=MATCH_THRESHOLD) if tracking else␣
↪None
        canvas.draw_image(frame, 0, 0)
        if result is not None:
            x, y, w, h, score = result
            message = "Tracking - BOOT: select"
            canvas.draw_rectangle(x, y, w, h, color=(0, 255, 0), thickness=2)
            canvas.draw_string(view[0] + 2, view[1] + visible_h - 14,
                               "match %.2f  %d ms" % (score, time.ticks_diff(time.
↪ticks_ms(), begin)),
                               color=(0, 255, 0))
        elif tracking:
            # Preserve the template and retry near the last accepted position.
            message = "Lost - searching..."
        else:
            message = "Aim box, press BOOT"
            canvas.draw_rectangle(roi, color=(255, 255, 0), thickness=2)
        canvas.draw_string(view[0] + 2, view[1] + 2, message, color=(255, 255,␣
↪255))
        lcd.write(canvas, x=offset_x, y=offset_y, roi=view, fit=False)
        time.sleep_ms(0 if tracking else 10)
finally:
    try:
        if tracker is not None:
            tracker.deinit()
    finally:
        if lcd is not None:
            lcd.deinit()
```

`init(image, roi)` caches target features, then frees the template model. Search and head models remain loaded; selecting a new target reloads the template model temporarily. Use unannotated RGB565 or grayscale images of the same dimensions throughout a track. The returned tuple is `(x, y, w, h, score)`; below-threshold results return `None` without updating the previous search state. Scores do not guarantee target visibility. This is local single-object tracking without automatic global reacquisition or training of network weights.

6.6.2 Raw Output Tensors

```
import sensor, espdl

model = espdl.Model("/sdcard/custom.espdl", mean=(0, 0, 0), std=(255, 255, 255),␣
↪letterbox=True)
try:
    print("inputs:", model.inputs())
    print("outputs:", model.outputs())
    img = sensor.snapshot()
    outputs = model.predict(img)
```

(continues on next page)

(continued from previous page)

```
    for name, tensor in outputs.items():
        _, shape, dtype, exponent, raw = tensor
        print(name, shape, dtype, exponent, len(raw))
finally:
    model.deinit()
```

Use `Model` when the built-in task wrappers do not match a model's output layout. ESP-DL still performs image preprocessing and inference; `predict()` returns the raw output tensors so Python code can run the model-specific decode. `inputs()` and `outputs()` return `TensorInfo` tuples `(name, shape, dtype, exponent)`. `predict()` returns `RawTensor` tuples `(name, shape, dtype, exponent, bytes)` keyed by output tensor name. Decode the raw bytes according to `dtype`, apply the ESP-DL power-of-two `exponent` scale, and run logic such as sigmoid, box decode, NMS, softmax, or top-k selection. When `letterbox=True`, remove the model-input padding before mapping coordinates back to the source image.

6.6.3 Object Detection

```
import sensor, espdl

det = espdl.ESPDet("/sdcard/espdet_pico_224_224_face.espdl", score=0.5, nms=0.7)
try:
    img = sensor.snapshot()
    for x, y, w, h, score, category in det.detect(img):
        img.draw_rectangle(x, y, w, h, color=(255, 0, 0), thickness=2)
        img.draw_string(x, max(0, y - 12), "%.2f:%d" % (score, category))
finally:
    det.deinit()
```

`score` removes low-confidence candidates, while `nms` controls suppression of overlapping boxes. Detection coordinates are mapped back to the input image and can be used directly for drawing. Load a model once and reuse it across frames; `deinit()` releases model weights and intermediate buffers.

6.6.4 PicoDet

PicoDet is the detector family used by the bundled pedestrian model. Its anchor-point output tensors are decoded by ESP-DL's Pico postprocessor; this output layout is distinct from ESPDet and YOLO11.

6.6.5 PedestrianDetect

`PedestrianDetect` is the Python API for the bundled 224x224 PicoDet pedestrian model. It uses the model's preprocessing defaults and returns the standard detection tuples:

```
det = espdl.PedestrianDetect(
    "/sdcard/pedestrian_detect/pedestrian_detect_pico.espdl",
    score=0.7,
    nms=0.5,
)
```

6.6.6 Image Classification

```
import espdl, image

img = image.Image("/sdcard/cat.jpg").to_rgb565(copy=True)
classifier = espdl.ImageNetCls(
```

(continues on next page)

(continued from previous page)

```
    "/sdcard/imagenet_cls_mobilenetv2_s8_v1.espdl",
    topk=5,
    score=0.0,
)
try:
    for label, score in classifier.classify(img):
        print(label, "%.4f" % score)
finally:
    classifier.deinit()
```

Classification returns up to `topk (label, score)` pairs ordered by the model output. Convert file images to RGB565 when necessary so the input format is accepted by the wrapper.

6.6.7 Pose Estimation

```
pose = espdl.YOLO11nPose("/sdcard/espdet_yolo11n_pose_160_160_coco.espdl", score=0.
↪35, topk=5)
try:
    img = sensor.snapshot()
    for x, y, w, h, score, category, keypoints in pose.detect(img):
        img.draw_rectangle(x, y, w, h, color=(255, 0, 0))
        for px, py in keypoints:
            if px > 0 and py > 0:
                img.draw_circle(px, py, 2, color=(0, 255, 0), fill=True)
finally:
    pose.deinit()
```

Each pose result contains 17 COCO keypoints. A missing or low-confidence point is returned as `(0, 0)` and should be skipped before drawing or computing joint geometry.

6.6.8 Adjust Thresholds at Runtime

```
det.set_thresholds(score=0.65, nms=0.6)
```

Thresholds can be changed without reloading the model, which is useful when adapting an application to lighting, distance, or scene-density changes.

6.6.9 Result tuples

- TensorInfo: `(name, shape, dtype, exponent)`
- RawTensor: `(name, shape, dtype, exponent, bytes)`
- Detection: `(x, y, w, h, score, category)`
- Pose: `(x, y, w, h, score, category, keypoints)` with 17 COCO keypoints
- Classification: `(label, score)`
- Tracking: `(x, y, w, h, score)` or `None` below the match threshold

See also:

AI Inference describes the ESP-DL inference pipeline, the `.espdl` format, quantization, and pre-/post-processing. To deploy a new model, see *Introduce a New Model*.

Runnable examples: `example/03-Machine-Learning/00-ESP-DL` (pedestrian detection, ESP-DL raw output decoding, ESPDet, YOLO11, pose, ImageNet classification, BOOT-confirmed NanoTrack tracking).

6.6.10 Functions

espdl.**load_model**(*path*, *, *profile=...*)

Preload an ESP-DL model file.

Parameters

- **path** –model path, for example "/sdcard/model.espdl" or "/flash/model.espdl".
- **profile** –True enables ESP-DL profiling output when supported.

6.6.11 Classes

class espdl.**NanoTrack**(*template*, *search*, *head*)

NanoTrack V2 single-object tracker using full-INT8 NHWC ESP-DL models.

Load search and head models. The template model is loaded only during init().

Parameters

- **template** –127x127 template backbone .espdl path.
- **search** –255x255 search backbone .espdl path.
- **head** –two-input NanoTrack V2 head .espdl path.

init(*image*, *roi*)

Cache target features from an unannotated RGB565/grayscale image, then release the template model.

Parameters

roi –(x, y, w, h), inside the image, at least 10x10 pixels.

track(*image*, *, *threshold=...*)

Track in a same-size image; return (x, y, w, h, score), or None below threshold.

Parameters

threshold –raw foreground score in [0, 1]; rejected frames do not update the search state.

deinit()

Release models and cached features. Repeated calls are safe.

class espdl.**Model**(*path*, *, *mean=...*, *std=...*, *rgb_swap=...*, *letterbox=...*, *pad=...*)

Generic ESP-DL model runner for examples that implement model-specific decoding in Python.

Create a raw model runner from an .espdl model.

Parameters

- **path** –model path.
- **mean** –optional RGB mean values for image preprocessing.
- **std** –optional RGB standard deviation values for image preprocessing.
- **rgb_swap** –True converts RGB input to BGR before model inference.
- **letterbox** –True keeps aspect ratio and pads the model input.
- **pad** –optional RGB padding values used when letterbox is enabled.

deinit()

Release model resources.

inputs()

Return model input metadata keyed by tensor name.

outputs()

Return model output metadata keyed by tensor name.

predict(*image*)

Run model inference on an image and return raw output tensors keyed by tensor name.

Parameters

image –RGB565 or grayscale image.

class espdl.**ESPDet**(*path*, *, *score*=..., *nms*=..., *mean*=..., *std*=...)

ESP-DL object detection wrapper for ESPDet models.

Create a detector from an .espdl model.

Parameters

- **path** –model path.
- **score** –optional confidence threshold.
- **nms** –optional non-maximum suppression threshold.
- **mean** –optional RGB mean values for preprocessing.
- **std** –optional RGB standard deviation values for preprocessing.

deinit()

Release model resources.

detect(*image*)

Run object detection on an image.

Parameters

image –RGB565 or grayscale image.

set_thresholds(*, *score*=..., *nms*=...)

Update detector thresholds.

Parameters

- **score** –new confidence threshold, or None to keep current value.
- **nms** –new NMS threshold, or None to keep current value.

class espdl.**PedestrianDetect**(*path*, *, *score*=..., *nms*=..., *topk*=...)

ESP-DL PicoDet pedestrian detection wrapper.

Create a pedestrian detector from the bundled ESP-DL model.

Parameters

- **path** –model path.
- **score** –optional confidence threshold; defaults to the upstream model's 0.7.
- **nms** –optional non-maximum suppression threshold; defaults to the upstream model's 0.5.
- **topk** –maximum number of detections returned per frame.

deinit()

Release model resources.

detect(*image*)

Run pedestrian detection on an RGB565 or grayscale image.

set_thresholds(*, *score*=..., *nms*=...)

Update detector thresholds.

Parameters

- **score** –new confidence threshold, or None to keep current value.
- **nms** –new NMS threshold, or None to keep current value.

class espdl.**YOLO11**(*path*, *, *score*=..., *nms*=..., *topk*=..., *mean*=..., *std*=...)

ESP-DL YOLO11 object detection wrapper.

Create a YOLO11 detector from an .espdl model.

Parameters

- **path** –model path.
- **score** –optional confidence threshold.
- **nms** –optional non-maximum suppression threshold.
- **topk** –maximum number of detections returned per frame.
- **mean** –optional RGB mean values for preprocessing.

- **std** –optional RGB standard deviation values for preprocessing.

deinit()

Release model resources.

detect(*image*)

Run object detection on an image.

Parameters

image –RGB565 or grayscale image.

set_thresholds(*, *score*=..., *nms*=...)

Update detector thresholds.

Parameters

- **score** –new confidence threshold, or None to keep current value.
- **nms** –new NMS threshold, or None to keep current value.

class espdl.**YOLO11nPose**(*path*, *, *score*=..., *nms*=..., *topk*=..., *mean*=..., *std*=...)

ESP-DL YOLO11n COCO pose wrapper.

Create a YOLO11n pose detector from an .espdl model.

Parameters

- **path** –model path.
- **score** –optional confidence threshold.
- **nms** –optional non-maximum suppression threshold.
- **topk** –maximum number of pose results returned per frame.
- **mean** –optional RGB mean values for preprocessing.
- **std** –optional RGB standard deviation values for preprocessing.

deinit()

Release model resources.

detect(*image*)

Run COCO pose detection on an image. Returns 17 COCO keypoints for each person.

Parameters

image –RGB565 or grayscale image.

set_thresholds(*, *score*=..., *nms*=...)

Update detector thresholds.

Parameters

- **score** –new confidence threshold, or None to keep current value.
- **nms** –new NMS threshold, or None to keep current value.

class espdl.**ImageNetCls**(*path*, *, *topk*=..., *score*=..., *mean*=..., *std*=..., *softmax*=...)

ESP-DL ImageNet classification wrapper.

Create a classifier from an .espdl model.

Parameters

- **path** –model path.
- **topk** –maximum number of classes returned.
- **score** –optional minimum score threshold.
- **mean** –optional RGB mean values for preprocessing.
- **std** –optional RGB standard deviation values for preprocessing.
- **softmax** –True applies softmax to output scores.

deinit()

Release model resources.

classify(*image*)

Run image classification on an image.

Parameters

image –RGB565 or grayscale image.

set_thresholds(*, *topk*=..., *score*=...)

Update classifier thresholds.

Parameters

- **topk** –new maximum result count, or None to keep current value.
- **score** –new minimum score, or None to keep current value.

6.7 tflite –Model Inference

The `tflite` module loads TensorFlow Lite `.tflite` flatbuffers and runs them through TensorFlow Lite Micro. It provides a general model execution interface for TFLite models, while higher-level task helpers can be layered on top when a model family needs shared preprocessing or post-processing.

6.7.1 Basic Usage

```
import tflite

model = tflite.Model("/sdcard/sine.tflite")
try:
    print("input:", model.input_shape, model.input_dtype, model.input_scale, model.
↪input_zero_point)
    print("output:", model.output_shape, model.output_dtype, model.output_scale,␣
↪model.output_zero_point)
    outputs = model.predict([input_array])
    raw = outputs[0]
finally:
    model.deinit()
```

`predict()` returns raw output tensors as ulab ndarrays. Quantized outputs are not automatically decoded; use `output_scale` and `output_zero_point` to convert values when the model requires it.

6.7.2 Callable Inputs

For image models it is often cheaper to fill the input tensor directly instead of building an intermediate ndarray:

```
def fill_input(buffer, shape, dtype_code):
    # Fill buffer with model-specific quantized bytes.
    ...

outputs = model.predict([fill_input])
```

The callable receives a writable bytearray view of the raw input tensor, the tensor shape, and an integer dtype code such as `ord("b")` for int8. This path is useful for image resizing, grayscale conversion, and quantization in Python examples.

6.7.3 Post-processing

`Model` can take a default `postprocess` callback, and `predict()` can take a one-shot `callback`. The callback receives `(model, inputs, raw_outputs)` and can return any Python object.

See also:

Runnable examples: `example/03-Machine-Learning/01-TFLite` (person/no-person and sine smoke tests).

6.7.4 Classes

class `tflite.`**Model**(*path*, *, *postprocess*=...)

TensorFlow Lite Micro model wrapper.

Load a .tflite flatbuffer from the filesystem.

Parameters

- **path** –model path, for example "/sdcard/model.tflite" or "/flash/model.tflite".
- **postprocess** –optional callback that receives (model, inputs, raw_outputs) and returns the final result.

deinit()

Release model, interpreter, and tensor arena resources.

predict(*inputs*, *, *callback*=...)

Run inference.

Parameters

- **inputs** –one item per input tensor. Each item is an ulab ndarray or a callable that fills the raw tensor buffer.
- **callback** –optional one-shot post-processing callback; overrides the default postprocess callback.

6.8 image.ImageIO –Image Stream

The `image.ImageIO` type records and replays sequences of images while preserving inter-frame timing. A file stream reads/writes a container on storage; a memory stream keeps frames in a pre-allocated buffer. It is exposed on the `image` module as `image.ImageIO`; the type stub lives in `stubs/imageio.pyi`.

6.8.1 Record to Storage

```
import sensor, image

stream = image.ImageIO("/sdcard/stream.bin", "w")
for _ in range(30):
    stream.write(sensor.snapshot())
stream.sync()
print("frames:", stream.count(), "bytes:", stream.size())
stream.close()
```

Each `write()` stores the image and the elapsed time since the previous frame. `sync()` flushes buffered file data without closing the stream. Always close a file stream so the container metadata and storage buffers are finalized.

6.8.2 Replay a Recorded Stream

```
import image

stream = image.ImageIO("/sdcard/stream.bin", "r")
try:
    while True:
        img = stream.read(loop=False, pause=True)
        if img is None:
            break
        img.flush()
finally:
    stream.close()
```

`pause=True` reproduces the recorded frame timing, while `loop=False` returns `None` at the end instead of rewinding. Set `pause=False` when frames should be consumed as quickly as processing permits.

6.8.3 Use an In-Memory Stream

```
stream = image.ImageIO((320, 240, image.RGB565), 10)
for _ in range(10):
    stream.write(sensor.snapshot())

stream.seek(0)
first = stream.read(pause=False)
print("buffer bytes:", stream.buffer_size())
stream.close()
```

Memory streams avoid storage I/O and are useful for short frame histories or frame-difference workflows, but their full capacity is allocated in RAM or PSRAM when the stream is created.

See also:

Codecs and Streaming covers ImageIO alongside JPEG, H.264, and the USB CDC preview path.

Runnable example: `example/02-Image-Processing/03-Frame-Differencing/in_memory_frame_differencing.py` uses an in-memory ImageIO stream.

6.8.4 Constants

`imageio.`**`FILE_STREAM`**

Stream backed by a file on storage (e.g. `/sdcard/stream.bin`).

`imageio.`**`MEMORY_STREAM`**

Stream backed by a fixed-size buffer in PSRAM/RAM.

6.8.5 Classes

`class` `imageio.`**`ImageIO`**(*stream*, *mode*)

`class` `imageio.`**`ImageIO`**(*stream*, *count*)

Record and replay sequences of images, preserving inter-frame timing. Exposed as `image.ImageIO`. A file stream reads/writes the OpenMV "OMV IMG STR" container on storage; a memory stream keeps frames in a pre-allocated buffer for fast capture/playback.

Open a file stream for reading ("r") or writing ("w").

Parameters

- **`stream`** –path to the stream file.
- **`mode`** – "r" to read, "w" to create/overwrite.

type()

Return the stream type: FILE_STREAM or MEMORY_STREAM.

is_closed()

Return True if the stream has been closed.

count()

Return the number of frames recorded in the stream.

offset()

Return the current frame index (read/write cursor).

version()

Return the container version for file streams, or None for memory streams.

buffer_size()

Return the per-frame buffer size for memory streams, or None for file streams.

size()

Return the total stream size in bytes.

write(*image*)

Append one image to the stream (records the elapsed time since the last write).

Parameters

image –frame to store.

read(*copy_to_fb=...*, *, *loop=...*, *pause=...*)

Read the next frame from the stream.

Parameters

- **copy_to_fb** –True loads the frame into the frame buffer (and updates the preview).
- **loop** –True rewinds to the first frame at end-of-stream instead of returning None.
- **pause** –True sleeps to honor the frame's recorded timestamp (real-time playback).

seek(*offset*)

Move the read/write cursor to the given frame index.

Parameters

offset –target frame index.

sync()

Flush buffered file data to storage (no-op for memory streams).

close()

Close the stream and release its resources.

6.9 h264 –H.264 Encoding

The `h264` module encodes images into Annex-B H.264 NAL units, for example to record video to storage or to feed the *rtsp –RTSP Streaming* server. It is available on ESP32-P4 builds.

6.9.1 Encode One Frame

```
import sensor, h264

enc = h264.H264Encoder(320, 240, fps=15)
nal = enc.encode(sensor.snapshot())
print("bytes:", len(nal), "keyframe:", enc.keyframe())
enc.close()
```

The encoder dimensions must match every input image. `encode()` returns one encoded frame as `bytes` and `keyframe()` reports whether the most recently encoded frame is an IDR/I frame.

6.9.2 Record a Raw H.264 Stream

```
import sensor, h264

FRAME_COUNT = 300
OUT_PATH = "/sdcard/out.h264"

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.skip_frames(time=1000)

first = sensor.snapshot()
enc = h264.H264Encoder(
    first.width(),
    first.height(),
    fps=15,
    bitrate=1_500_000,
    gop=15,
)

try:
    with open(OUT_PATH, "wb") as output:
        output.write(enc.encode(first))
        for _ in range(FRAME_COUNT - 1):
            output.write(enc.encode(sensor.snapshot()))
finally:
    enc.close()
```

This writes an elementary Annex-B stream without an MP4 container. It can be played with `ffplay out.h264` or remuxed on a host. `bitrate` is the target number of bits per second, `gop` is the keyframe interval in frames, and the QP bounds control the permitted quality/size range.

6.9.3 Resource and Throughput Considerations

Create one encoder for a fixed resolution and reuse it throughout the stream. Close it when finished. If capture or storage cannot sustain the configured frame rate and bitrate, reduce the resolution, frame rate, or bitrate instead of repeatedly recreating the encoder.

See also:

Codecs and Streaming explains the encode-then-stream pipeline and how H.264 keyframes interact with RTSP delivery.

Runnable example: `example/01-Camera/01-H264/record_h264.py`.

6.9.4 Classes

class h264.**H264Encoder**(*width*, *height*, *, *fps=...*, *gop=...*, *bitrate=...*, *qp_min=...*, *qp_max=...*)

Hardware-accelerated H.264 video encoder. Feed it images frame by frame with encode(); it returns Annex-B NAL units ready to mux into a file or stream over the network (see the rtsp module).

Open an encoder for a fixed frame size.

Parameters

- **width** –frame width in pixels.
- **height** –frame height in pixels.
- **fps** –target frame rate; also the default GOP length.
- **gop** –keyframe (IDR) interval in frames; 0 selects one keyframe per second (== fps).
- **bitrate** –target bitrate in bits per second; 0 auto-selects width*height*fps.
- **qp_min** –lower quantization-parameter bound (better quality, larger frames).
- **qp_max** –upper quantization-parameter bound (lower quality, smaller frames).

encode(*image*)

Encode one image and return its Annex-B NAL units as bytes.

Parameters

image –source frame; its size must match the encoder width/height.

keyframe()

Return True if the most recently encoded frame was a keyframe (IDR/I).

close()

Release the encoder. Using the object afterwards raises OSError.

6.10 rtsp –RTSP Streaming

The rtsp module serves H.264 NAL units over RTSP so a client such as VLC or ffplay can view the camera stream over the network. Pair it with *h264 –H.264 Encoding*. It is available on ESP32-P4 builds.

6.10.1 Capture, Encode, and Stream

```
import network, sensor, h264, rtsp, time
from machine import Pin

WIDTH, HEIGHT, FPS = 320, 240, 30

lan = network.LAN(
    mdc=Pin(31),
    mdio=Pin(52),
    reset=Pin(51),
    phy_addr=1,
    phy_type=network.PHY_IP101,
)
lan.active(True)

for _ in range(100):
    if lan.isconnected():
        break
    time.sleep_ms(100)
if not lan.isconnected():
    raise OSError("Ethernet connection failed")

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
```

(continues on next page)

(continued from previous page)

```
sensor.set_framesize(sensor.QVGA)
sensor.skip_frames(time=1000)

enc = h264.H264Encoder(WIDTH, HEIGHT, fps=FPS, bitrate=3_000_000)
server = rtsp.RTSPServer(WIDTH, HEIGHT, fps=FPS, listen_port=8554)

print("rtsp://%s:8554/" % lan.ifconfig()[0])
try:
    while True:
        server.send(enc.encode(sensor.snapshot()))
finally:
    server.stop()
    enc.close()
```

The Ethernet pin mapping above is for the ESP32-P4X-Function-EV-Board with its IP101 RMII PHY; use the network configuration appropriate for another product board. The width, height, and frame rate advertised by `RTSPServer` must match the encoder configuration.

6.10.2 Client and Queue Behavior

Open the printed URL with `ffplay rtsp://<board-ip>:8554/` or an RTSP-capable player. `send()` queues one encoded frame and does not wait for a client; when no client is connected or the client is too slow, frames may be dropped so the capture loop is not blocked. `max_frame_len` can be increased when high-resolution or high-bitrate encoded frames exceed the default limit.

6.10.3 Shutdown

Always call `server.stop()` before releasing network or encoder resources. A `try/finally` block ensures cleanup also occurs when the script is interrupted or an exception is raised.

See also:

Codecs and Streaming covers the full capture, encode, and stream pipeline.

Runnable example: `example/01-Camera/02-RTSP/stream_rtsp.py`.

6.10.4 Classes

class rtsp.**RTSPServer**(*width*, *height*, *, *fps=...*, *listen_port=...*, *max_frame_len=...*)

Minimal RTSP server that streams H.264 video over the network. Create it once Wi-Fi is up, then push encoded NAL units from an h264.H264Encoder with send(). Clients connect to `rtsp://<board-ip>:<listen_port>/`. Only video (no audio) is served.

Start the RTSP server and advertise the given stream format.

Parameters

- **width** –advertised video width in pixels.
- **height** –advertised video height in pixels.
- **fps** –advertised frame rate, used for SDP and pacing.
- **listen_port** –TCP port the RTSP service binds to.
- **max_frame_len** –max accepted encoded frame size in bytes; 0 uses width*height as a safe ceiling.

send(*nal*)

Queue one encoded H.264 frame (Annex-B NAL units) for delivery. Frames are sent in order; if the client is slow or absent the frame is dropped rather than blocking the caller. Frames larger than max_frame_len are ignored.

Parameters

nal –encoded frame bytes from h264.H264Encoder.encode().

stop()

Stop the server, join its task, and free queued frames.

6.11 uvc —USB webcam output

Boards with USB-OTG device support expose CDC, MSC and UVC together at startup. Open the ESP-VISION Camera in a host webcam application. The default image is the built-in ESP-VISION logo; opening UVC does not initialize or reconfigure `sensor`. Boards that disable USB device support do not provide this module.

The initial mode is MJPEG, 320×240, 10 FPS with isochronous transport. JPEG frames must fit within 48 KiB. Actual delivery depends on host scheduling and concurrent USB traffic. `get_mode()` reports the supported mode; `is_streaming()` reports whether the host enabled the streaming interface, not whether a particular application displayed a frame.

`uvc.`**write**(*img*)

Publish a processed QVGA `image.Image`. Returns `False` if the host is not streaming and `True` once the frame has been copied into firmware-owned storage. Acceptance does not acknowledge host receipt. RGB565, grayscale and JPEG inputs use the shared JPEG encoder. Invalid dimensions or oversized encoded frames raise `ValueError`; encoding/allocation failures raise `OSError`. Camera buffers may be reused immediately after return. New pending frames replace older pending frames; an in-flight frame is never overwritten.

`uvc.`**set_frame**(*img*)

Copy a fixed QVGA image even without an open host. Detach the preview source and repeat the copied frame until replaced or reset. Returns `True` on success. Failed updates preserve the previous source.

`uvc.`**bind**(*source*)

Bind `"preview"` to the processed images published by `img.flush()`. UVC and EV-MUX share one JPEG encode and have independent transport readiness. UVC works without an IDE session. Images of other dimensions are ignored by UVC while remaining eligible for EV-MUX. Oversized frames or UVC allocation failures are dropped without blocking EV-MUX. `None` detaches and restores the logo. Binding also starts from the logo until a matching frame is published.

`uvc.`**reset**()

Detach the source and restore the logo. This does not reset the USB controller or disconnect CDC/MSC. VM soft reset also restores the logo. A stopped or idle live producer otherwise leaves its last frame repeating.

`uvc.`**is_streaming**()

Return whether the host has enabled streaming and the USB bus is not suspended.

`uvc.`**get_mode**()

Return `{"format": "MJPEG", "width": 320, "height": 240, "interval_us": 100000}`. Only this mode is advertised and accepted.

A processed camera application can publish to both outputs:

```
import sensor
import uvc

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
uvc.bind("preview")
try:
    while True:
        img = sensor.snapshot()
```

(continues on next page)

(continued from previous page)

```
        img.draw_rectangle((20, 20, 100, 80), color=(255, 0, 0))
        img.flush()
finally:
    uvc.reset()
```

Direct `uvc.write(img)` is also available when EV-MUX preview is unnecessary. Do not call both publishing paths for the same frame. No Python callback or camera capture executes inside USB callbacks.

6.12 storage –Diagnostics and Confirmed Recovery

Inspect a volume without modifying it, retry mounting without formatting, or explicitly confirm recovery of an unmounted Flash `ffat`. Storage failures return diagnostic results; invalid arguments raise Python exceptions. See *Storage, Filesystems and Data Protection* for the complete data-protection, backup and recovery contract. A successful recovery may require a user-selected soft reset to publish the Python mount.

Online SD formatting and formatting a mounted volume are not supported. Confirmation must be the literal boolean `True` after the user has approved erasing the selected volume; callers must not set it automatically after an error.

6.12.1 Functions

`storage.`**`status`**`(`*volume*=...`)`

Return non-destructive status and recovery guidance for flash or sd. Storage faults are represented in the result, not by rebooting the board.

Parameters

`volume` – "flash" (default) or "sd".

`storage.`**`retry`**`(`*volume*=...`)`

Retry mounting without formatting or erasing any data, including when board policy disables automatic mounting. Check ok and operation_error; after success, soft_reset_required requests an explicit soft reset to publish the Python mount.

Parameters

`volume` – "flash" (default) or "sd".

`storage.`**`format`**`(`*volume*=..., *, *confirm*=...`)`

Explicitly initialize an unmounted flash data/FAT partition named ffat. This destroys files and requires the literal confirm=True after user approval and any required backup. Mounted volumes, legacy vfs, open handles, quarantined writers and online SD formatting are not supported. Check ok, operation_error and soft_reset_required; failure never triggers automatic reboot or another format.

Parameters

- **`volume`** – "flash" (default); "sd" returns an unsupported operation result.
- **`confirm`** –Must be the boolean True; omission, False and truthy strings are rejected before any write.

Chapter 7

How-To Guides

Task-oriented guides for extending ESP-VISION.

7.1 Build vision applications with AI

Describe an idea to your AI assistant and develop it into a Python script running on an ESP-VISION board. The assistant can find examples, prepare firmware and models, deploy the application, inspect camera images and logs, and improve the result with you.

7.1.1 Firmware-matched knowledge

The plugin reads the selected board's actual firmware version before querying APIs, capabilities or examples. The online Knowledge service retains historical snapshots and requires an exact firmware version for every content query. Updating the plugin does not require updating the firmware; an older board can continue using its matching snapshot.

After flashing or switching boards, the assistant inspects the device again. If the firmware version is missing or no matching snapshot exists, it explains the limitation and offers an official firmware update. If you explicitly choose not to update, it uses the latest available official firmware Knowledge snapshot, tells you the selected version and warns that its APIs may differ from your device. The official catalog designates the latest firmware; the assistant preserves its exact TAG instead of guessing order from its spelling. It waits for your choice and never silently switches versions. A failed update or unavailable catalog does not trigger fallback. This choice is reset after switching devices, reconnecting or flashing. Development-version suffixes are preserved. For offline questions, state the target firmware version explicitly, for example: "Explain camera capture for firmware 2026.10.21." The assistant can list supported versions before proceeding.

7.1.2 Connect your AI assistant

The ESP-VISION plugin connects your assistant to official knowledge and local device tools. Choose the setup path for your client in the official getting-started guide.

WorkBuddy and QoderWork

Install through the application marketplace: search for `esp-vision` and select the listing below. These are the listing titles shown by the official website.

Table 1: Marketplace listings

Client	Listing
WorkBuddy	嵌入式视觉开发专家	ESP-Vision AI 视觉
QoderWork	ESP-Vision 嵌入式 AI 视觉开发

Codex and Claude Code

For Codex, start with:

Note: Help me use ESP-VISION in Codex. Official installation guide: https://www.npmjs.com/package/@esp-vision/plugin

---

For Claude Code:

Note: Help me use ESP-VISION in Claude Code. Official installation guide: https://www.npmjs.com/package/@esp-vision/plugin

---

Cursor and other assistants

For Cursor and other clients, see the plugin installation guide.

Note: I want to use ESP-VISION in my current AI assistant. Official setup guide: https://github.com/espressif/esp-vision/blob/master/host/esp-vision-plugin/README.md

---

After setup, start a new conversation in your project. For USB development, the AI assistant needs access to the computer connected to the board.

7.1.3 Connect a board and run an application

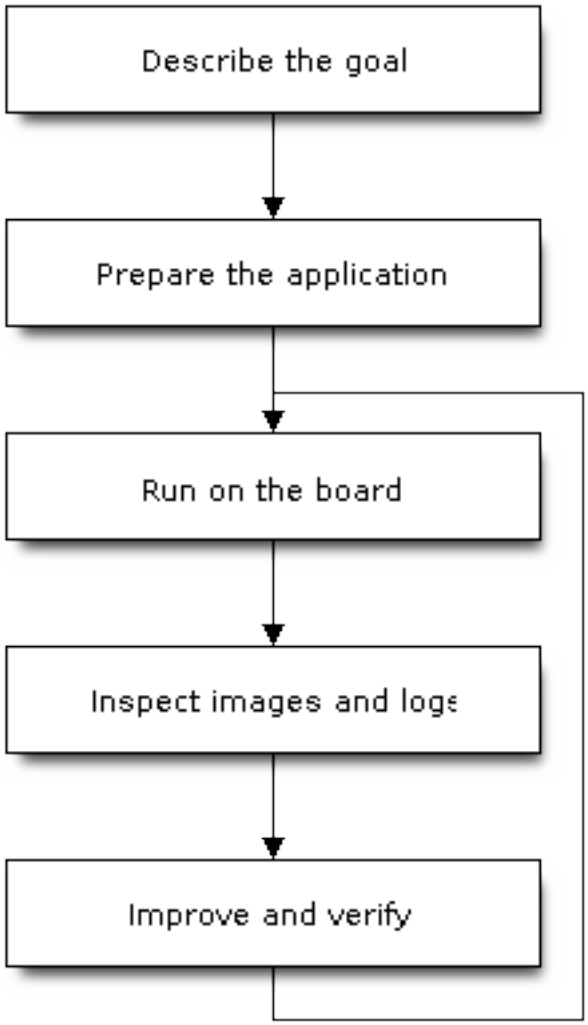
Connect a *board supported by ESP-VISION* to your computer with a USB data cable. For first use, the assistant can download and flash official firmware to prepare the board for vision applications:

Note: This is a new board supported by ESP-VISION. Install ESP-VISION firmware so I can take pictures with its camera. There is no data on the board that I need to keep.

After the initial firmware installation, start developing an application. A board that already has compatible firmware can begin with the application request; a device with networking configured can also connect over the local network.

Note: Show live camera video on the board’s screen with a frame counter, and leave it running for me to see.

Once running, the screen shows a continuously updating camera view and frame counter. You also receive the Python script to adapt and reuse.



7.1.4 Everyday tasks

Start with the feature, use case or problem you have in mind. The plugin's workflow handles implementation, while you can continue shaping the result.

Color tracking

Note: Track red objects in the camera view and draw a box around them in real time. The box should follow the object as it moves.

Move a red object in front of the camera and see how well the box follows it. You can add requirements for a different color, object or movement speed.

Update firmware

Note: Update the board to the latest compatible official ESP-VISION firmware, keeping my scripts and settings.

After the update, you receive the installed version and the device' s running results.

Learn about the connected device

Note: What model is the board connected to my computer, what firmware does it have, and which vision features does it support?

Device information helps you choose an application direction, such as image processing, object detection or a camera display.

Face detection

Note: Have the board detect faces and draw boxes around them on screen in real time, with the detection frame rate displayed.

Observe what happens when a face enters, moves through and leaves the camera view, and use the displayed frame rate to assess responsiveness.

QR-code recognition

Note: I want to read QR codes with the camera. Outline each code on screen, show its content and keep updating as I move it.

Add requirements for your use case, such as reading smaller QR codes or making their contents easier to read.

Develop over Wi-Fi

Note: Connect my board to Wi-Fi so I can update its program and view the camera over a wireless connection.

For initial setup, connect a Wi-Fi-capable board over USB and provide network details when the assistant asks. Once configured, continue development over the LAN; the board reconnects to its saved network after power-on. Alternatively, open the browser flashing panel on the website, choose "Configure Wi-Fi Only" under "Flash Mode" , connect the serial port, and enter and save your network settings. See *Host integration, Wi-Fi configuration and recovery* for the complete setup procedure and connection troubleshooting.

USB camera

Note: Turn the board into a USB camera so my computer can show live video with boxes around detected faces.

Use a board with USB-OTG Device support and connect its USB-OTG port. Select `ESP-VISION Camera` in your computer's camera application to view the result. See *uvc — USB webcam output* for supported image formats, resolution and APIs.

Investigate and fix a problem

Note: The board's screen goes black after the camera program starts. Please fix it.

Share what you observed, along with any available error message or image. You can also describe how the application behaved before the problem appeared.

7.1.5 Improve the result through conversation

Keep working in the same conversation so the assistant can use the project, device and previous results as context. For the tracking application, you could follow up with:

Note: The red-object tracking box flickers in dim lighting. I want the tracking to stay stable in low light.

For the detection application:

Note: The status text is too small. Make it larger and add the detection frame rate.

Describe the next goal in terms of your experience: steadier tracking, faster response, clearer images or more readable text.

7.1.6 Tools behind the conversation

The host-side tools connect natural-language development to official knowledge and real hardware:

Table 2: ESP-VISION host tools

Tool	Role
Plugin (`@esp-vision/plugin`)	Gives the AI client the ESP-VISION Skill, bundled Device CLI and official knowledge connection.
Device CLI (`@esp-vision/device-cli`)	Lets the assistant discover devices, manage firmware and files, run Python scripts and capture images on the local host.
Knowledge MCP	Supplies official API documentation, board capabilities and examples for the assistant to consult.

For further detail, see the *Device CLI reference*, *Python script guidance* and *host SDK guide*.

7.2 Add a New Python Module

ESP-VISION exposes Python modules through the MicroPython `USER_C_MODULES` mechanism. The binding layer lives in `modules/` and only does object conversion plus light API adaptation; heavy logic belongs in pure C or in `platform/`. This guide adds a module named `foo`.

7.2.1 Overview

```
modules/py_foo.c            # binding + MP_REGISTER_MODULE(MP_QSTR_foo, ...)
modules/qstrdefs_esp_vision.h   # any feature-gated qstrs
micropython.cmake           # add py_foo.c to the source list
stubs/foo.pyi               # type stub for IDE completion
docs/.../api-reference/foo.rst  # reference page
```

7.2.2 1. Create the Binding Source

Add `modules/py_foo.c`. Define the module's functions, build a globals dict, and self-register the module. Existing modules are the best templates; for a simple function-only module follow `modules/py_sensor.c`, and for a type/class module follow `modules/py_display.c`.

```
/*
 * SPDX-FileCopyrightText: 2026 Espressif Systems (Shanghai) CO LTD
 *
 * SPDX-License-Identifier: Apache-2.0
 */
#include "py/runtime.h"

static mp_obj_t foo_hello(void) {
    return mp_obj_new_int(42);
}
static MP_DEFINE_CONST_FUN_OBJ_0(foo_hello_obj, foo_hello);

static const mp_rom_map_elem_t foo_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_foo) },
    { MP_ROM_QSTR(MP_QSTR_hello), MP_ROM_PTR(&foo_hello_obj) },
};
static MP_DEFINE_CONST_DICT(foo_module_globals, foo_module_globals_table);

const mp_obj_module_t mp_module_foo = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t *)&foo_module_globals,
};

MP_REGISTER_MODULE(MP_QSTR_foo, mp_module_foo);
```

C++ modules use `.cpp` (see `modules/py_espdl.cpp`); the build already sets `-std=gnu++2b` for C++ sources.

7.2.3 2. Register qstrs If Needed

Most qstrs (`MP_QSTR_foo`, `MP_QSTR_hello`) are collected automatically from the source. Only add entries to `modules/qstrdefs_esp_vision.h` for names that are board-feature-gated or otherwise not seen by the qstr scanner, using the `Q(name)` form.

7.2.4 3. Wire the Source into the Build

Add the source to `ESP_VISION_MODULE_SOURCES` in `micropython.cmake`:

```
set(ESP_VISION_MODULE_SOURCES
    ${ESP_VISION_ROOT}/modules/py_display.c
    ${ESP_VISION_ROOT}/modules/py_image.c
    ${ESP_VISION_ROOT}/modules/py_foo.c    # new
```

(continues on next page)

(continued from previous page)

```
    ...
)
```

If the module is only valid on some chips, add it inside the matching `IDF_TARGET` block (the H.264 and RTSP modules are gated to `esp32p4` this way). If it needs an extra component, link it via `target_link_libraries` like `idf::zxing`.

7.2.5 4. Add a Type Stub

Create `stubs/foo.pyi` describing the public surface so IDEs can complete it. Keep the Apache SPDX header and match the style of the existing stubs.

7.2.6 5. Document the Module

Add `docs/en/api-reference/foo.rst` and `docs/zh_CN/api-reference/foo.rst` using the Python-domain directives (`.. py:module::`, `.. py:function::`, `.. py:class::`), and add `foo` to the `toctree` in `api-reference/index.rst` for both languages.

7.2.7 6. Build and Verify

```
idf.py --board ESP32_P4X_EYE build
```

Then check from the REPL:

```
import foo
print(foo.hello())
```

7.3 Contribution and Review

Open a pull request (PR) against the appropriate branch in ESP-VISION on GitHub. Describe the change, its tests, and any hardware used. Follow the repository's copyright and license rules, and add an entry under `Unreleased` in `CHANGELOG.md`. Maintainers review the PR on GitHub and may request changes there.

7.3.1 Sync and merge process

1. After review, the maintainer comments `sha=<commit>` with the current PR head SHA and applies `PR-Sync-Merge` or `PR-Sync-Rebase`. The comment and label must come from the same maintainer. The current action rejects PRs that modify paths containing `.github` or `.gitlab`; coordinate those changes with maintainers.
2. The action creates a GitLab MR. If the contributor updates the PR, the maintainer comments the new head SHA and applies or reapplies `PR-Sync-Update` while retaining the original merge or rebase label.
3. Maintainers review the GitLab MR and run the applicable CI and hardware checks. They communicate requested changes on the GitHub PR. Once approved, they merge the GitLab MR into its target branch.
4. GitLab CI publishes the tested branch to GitHub. Maintainers verify that the PR shows the correct status and link the resulting commit; if rewritten history prevents automatic recognition, they close the PR with an explanation.

The workflow depends on GitHub secrets and network access to the internal GitLab. A failed or pending sync does not mean that the contribution has been merged. No Jira issue or Jira project is required.

7.4 Customize Firmware Features

ESP-VISION separates application-facing modules, image algorithms, standard MicroPython features, frozen Python code, and board services into independent configuration layers. This lets a product firmware remove unused capabilities to reduce flash, RAM, dependencies, and attack surface, or add product-specific modules without rewriting the camera and media pipeline.

7.4.1 Choose the Customization Scope

Before editing configuration, decide whether the change belongs to every ESP-VISION firmware or only one board. Changes to the root `micropython.cmake` affect every board whose chip matches the edited condition. Changes under `boards/<BOARD>/` (including the MicroPython-port files in `boards/<BOARD>/port/`) are board-specific. For a product variant, create a dedicated board package as described in *Add a New Board* instead of changing a shared development-board profile.

7.4.2 Customize ESP-VISION Python Modules

The `ESP_VISION_MODULE_SOURCES` list in `micropython.cmake` defines the C/C++ bindings exposed to Python. Remove a source only together with any helper source it requires, or append a new source as described in *Add a New Python Module*. Chip-specific modules belong inside an `IDF_TARGET` condition; for example, H.264 and RTSP are currently enabled only for `esp32p4`.

```
set(ESP_VISION_MODULE_SOURCES
    ${ESP_VISION_ROOT}/modules/py_display.c
    ${ESP_VISION_ROOT}/modules/py_image.c
    ${ESP_VISION_ROOT}/modules/py_imageio.c
    ${ESP_VISION_ROOT}/modules/py_helper.c
    ${ESP_VISION_ROOT}/modules/py_sensor.c
)
```

Removing a binding does not automatically remove every platform service or managed component behind it. After changing the source list, inspect `target_sources`, `target_link_libraries`, component manifests, and the final size report to confirm that the unwanted dependency is no longer linked.

7.4.3 Customize Image Algorithms

Each board's `boards/<BOARD>/imlib_config.h` selects optional `imlib` algorithms. Remove an `IMLIB_ENABLE_*` definition to omit an unused algorithm, or add a supported definition to expose another algorithm. Common groups include filters, geometry detection, QR codes, and AprilTags.

```
#define IMLIB_ENABLE_MEAN
#define IMLIB_ENABLE_GAUSSIAN
#define IMLIB_ENABLE_QRCODES
#define IMLIB_ENABLE_APRILTAGS
```

Some Python methods remain present in the binding even when their backend is disabled and will raise an exception at runtime. Keep the API documentation, examples, and generated board support table aligned with the selected algorithms. Do not remove `OMV_NO_GPL` without completing a license review.

7.4.4 Customize Standard MicroPython Features

Use `boards/<BOARD>/port/mpconfigboard.h` to override standard MicroPython feature macros for one board, such as networking, Bluetooth, ESP-NOW, ADC, SD card, USB, or other `MICROPY_PY_*` and `MICROPY_HW_*` options. Use `mpconfigboard.cmake` for CMake-level options and the board's sdkconfig chain.

```
#define MICROPY_PY_BLUETOOTH (0)
#define MICROPY_PY_ESPNOW (0)
#define MICROPY_PY_NETWORK_WLAN (1)
```

These macros can depend on ESP-IDF versions and SoC capability macros. A disabled Python feature may also require removing its ESP-IDF configuration or managed dependency before it produces a measurable firmware-size reduction.

7.4.5 Configure Memory Placement

PSRAM boards use the board's `sdkconfig` chain under `boards/<BOARD>/port/` to set the allocation boundary. `CONFIG_SPIRAM_MALLOC_ALWAYSINTERNAL` is the byte threshold used by ESP-IDF's default allocator and ESP-VISION's automatic allocator policy: allocations up to the threshold prefer internal RAM, while larger allocations prefer PSRAM. `CONFIG_SPIRAM_MALLOC_RESERVE_INTERNAL` separately reserves internal RAM for drivers and other allocations that require it. Adjust these values in the board's sdkconfig defaults when changing the memory balance.

Large image, model, frame, and scratch buffers prefer PSRAM when their required capabilities allow it. When a caller requires direct DMA or executable access, allocator fallbacks retain that capability. The P4X_EYE SPI display can also use an ordinary 8-bit framebuffer as a draw source because the LCD transfer layer stages it into bounded DMA transfers; this explicit copy path does not weaken direct-DMA requirements. Boards without PSRAM use internal memory where the capability is available.

7.4.6 Customize Frozen Python Code

The board's `boards/<BOARD>/manifest.py` controls Python modules frozen into firmware. Add product startup code, libraries, or packages with `freeze()`, `module()`, `package()`, or `include()` as supported by the MicroPython manifest system. Remove an entry when that frozen module is not required.

```
freeze("$(PORT_DIR)/modules")
freeze("$(ESP_VISION_ROOT)/modules", "py_inisetup.py")
freeze("$(ESP_VISION_ROOT)/boards/<BOARD>", "board_inisetup.py")
include("$(MPY_DIR)/extmod/asyncio")
```

Freezing code improves deployment consistency and startup availability but consumes firmware flash. Files intended to remain replaceable during development or after deployment should stay on the root filesystem, such as packages under `/lib`, or on `/sdcard` instead.

7.4.7 Customize Board Services and Optional Components

Board-specific camera, display, and SD card implementations live under `boards/<BOARD>/`. `micropython.cmake` automatically selects `camera.c`, `display.c`, and `sdcard.c` when present. Use `boards/<BOARD>/board.cmake` for board-level CMake switches; for example, current P4 boards enable the optional ZXing barcode backend with `ESP_VISION_ENABLE_BARCODE`.

```
set(ESP_VISION_ENABLE_BARCODE OFF)
```

When adding a component, register its source or IDF component and link it to `usermod_esp_vision_platform`. When removing one, remove the source, include path, compile definition, link dependency, and component-manifest entry as a single change.

7.4.8 Build and Verify

Reconfigure after changing CMake, sdkconfig, manifests, or feature macros, then build the selected board:

```
idf.py --board <BOARD> reconfigure
idf.py --board <BOARD> build
idf.py --board <BOARD> size
```

Verify imports and affected APIs on the REPL, run the relevant examples, and compare the size report before and after the change. Also rebuild the documentation because module navigation and board capability summaries are generated from the firmware configuration.

7.5 Publish qualified firmware

Firmware and Device Core have independent release schedules. Publishing an npm package, including Device Core 0.3, does not run the firmware upload job. Only a firmware tag matching the release CI rules runs `pack_launchpad`, `qualify_firmware_release` and `upload-dl-trigger`. Existing releases such as `2026.08.27` predate metadata generation: their binary downloads can work while `firmware/<board>/<version>.json` returns HTTP 404.

7.5.1 Package and URL contract

The packer reuses the release matrix's IDF artifacts; it does not rebuild firmware. Its `binaries/` artifact contains:

```
config.toml
releases/<tag>/manifest.json
esp-vision-<board>-<tag>.bin
esp-vision-<board>-<tag>-app.bin
firmware/<board>/<tag>.json
```

The uploader must keep `config.toml`, `releases/` and `firmware/` at `https://dl.espressif.com/AE/esp-vision/` and put both binary files in `bin/`. Core resolves metadata relative to the root config URL. Putting metadata in `bin/firmware/` causes HTTP 404 even though generation succeeded. The uploader accepts only the declared release objects and optional MCP files; unrelated artifacts are rejected.

For eight published boards, the default layout contains 16 BINs, eight per-board JSONs, one immutable manifest and one root config: 26 objects. The optional versioned MCP pack and compatibility `bin/mcp/latest.json` alias make 28. Single-bin mode removes eight application BINs and eight per-board JSONs: eight BINs plus manifest/config, or 10 objects (12 with MCP). Source configuration remains one TOML; the server keeps one canonical mutable firmware pointer inside that root config and immutable versioned manifests. The MCP latest alias is a separate compatibility convenience.

After generation, `tools/ci/launchpad/validate_release.py <tag>` checks every published board, including boards hidden from the website. It rejects absent application/install metadata, wrong board/version/build identity, incorrect URLs, corrupt binaries, incorrect bootloader/application/partition hashes, invalid silicon revision intersections and invalid partition bounds. It requires metadata for the current release only; older entries in the release list need not have metadata. The release job fails before handing an incomplete package to the uploader.

7.5.2 Immutable manifest and current pointer

Each version has one immutable `releases/<tag>/manifest.json` containing `schemaVersion = 1`, the version, full 40-character `sourceCommit`, positive CI `pipelineId`, `layout`, all board release descriptors, that version's complete `website` snapshot and an explicit `mcp` descriptor or JSON `null`. Model download URLs use the source commit, not a moving branch. Root `config.toml` keeps the launchpad fields and compatibility website snapshot, plus `[publication]` with the same identity, `manifestUrl`, SHA-256 of the exact manifest bytes, `previousConfigSha256` and `releases` records. Each record names only its versioned manifest/hash and actual nonempty unique `boards` set.

The packer downloads the previous public config and verifies every referenced historical manifest's hash, identity and board membership before carrying its record forward unchanged. Fetch or validation errors stop packaging. Git tags are not evidence of a successful publication and are never promoted into records. Legacy website version labels remain download-only; they do not acquire invented manifests or board/version combinations. A published version cannot be repackaged under the same name.

`manifest.website` is the authoritative versioned board/model snapshot. Root `config.website` mirrors the newest snapshot for compatibility. Consumers must follow the publication record for historical versions instead of combining today's board list with every version label.

7.5.3 Build-derived Wi-Fi capability

Every new release entry includes `capabilities.wifi` from `release-build-identity.json`. CI runs the real network-service compiler command in preprocessing-only mode after the build, using the actual include paths and definitions. Wi-Fi provisioning requires `MICROPY_PY_NETWORK_WLAN && ESP_VISION_ENABLE_BACKGROUND_NETWORK && (CONFIG_ESP_WIFI_ENABLED || CONFIG_ESP_WIFI_REMOTE_ENABLED)`. The identity also binds the board, firmware version, full source commit and application SHA-256. Missing or mismatched identity prevents qualification; presentation data and a board-name allowlist cannot claim Wi-Fi support.

7.5.4 Optional single-bin layout

`ESP_VISION_PUBLICATION_LAYOUT=legacy` remains the default. Enable `single-bin` explicitly only after the manifest/slicing-capable Core version and website consumer are deployed. It emits the eight complete binaries and central manifest, without application-only binaries or per-board JSON. Each application's `image.source` carries the complete container's `url`, `bytes` and `sha256` plus the application's byte `offset` within it. `image.url` equals the container URL; `image.bytes` and `image.sha256` still describe only the application. The source container must exactly match `install` and its slice offset must equal the application flash offset. Core verifies the complete container and application slice before writing only that application for an update. Complete installation retains its separate layout/security checks.

7.5.5 MCP and release qualification

MCP generation and schema validation precede manifest assembly. Its versioned descriptor binds the current pack's bytes/hash, version and full source commit. Failure blocks publication by default. `ESP_VISION_ALLOW_MISSING_MCP=1` explicitly permits `mcp: null`; the manifest must never point to an old `latest.json`. Compatibility alias maintenance is an uploader operation after the canonical publication has succeeded, not an input to qualification.

Packaging reuses successful firmware artifacts and can overlap hardware checks. `qualify_firmware_release` waits for the complete required firmware matrix, all three Function-board HIL variants, SDK checks and host/config/MCP checks, then validates the entire package with the actual commit and pipeline ID. The upload trigger needs both package and qualification success. The uploader independently verifies those jobs, stages and verifies immutable assets/manifests, and updates the mutable config pointer last with a compare-and-swap against `previousConfigSha256`. A stale/concurrent candidate must not overwrite newer history.

7.5.6 Release procedure

1. Merge the new uploader and Core contracts before enabling the new publication flow. Keep the companion root-directory fix as a prerequisite. Do not enable single-bin mode for legacy consumers.
2. Before tagging, separately bump the Plugin and Knowledge MCP npm versions and set matching Core/CLI minor ranges and resolved versions in both package manifests. The resolved CLI must already be published with the resolved Core bundled; the Plugin bundles that published CLI. Patch releases only fix bugs compatibly, only minor releases may add compatible features, and only major releases may break compatibility. Patch and

major releases do not add unrelated new features. Core and CLI versions publish independently from protected default-branch pipelines.

3. Wait for the exact source commit's GitHub mirror before creating a new firmware tag through the normal protected release process; commit-pinned model links must already resolve. The uploader rejects an unavailable source commit before advancing the config. Wait for build, HIL, SDK and package qualification; an npm publication does not publish firmware.
4. Verify the root config's publication hash against the versioned manifest, its exact board membership and website/MCP snapshots, and all selected image hashes. A dry-run config or a passed branch pipeline does not prove a production upload.

Historical binaries and immutable manifests remain available. Do not manufacture metadata for old images, replace an existing version, or guess compatibility from git tags. Offline tests and merging these changes do not publish a tag, npm package or CDN data.

7.5.7 Iris firmware files

`ESP_VISION_ENABLE_IRISFW` is an opt-in CMake option and C compilation macro, supported only on `ESP32_S31_MOSAICO`. Enabling it for another board fails configuration. The default remains OFF: ordinary Launchpad BIN builds keep their existing factory layout. The separate `firmware_irisfw_mosaico` job enables it with IDF `release-v6.1` and stores artifacts in `build/ESP32_S31_MOSAICO/idf6.1-irisfw`. Firmware tags package only this job through `pack_irisfw`; no other board produces an Iris package.

Use a new build directory when changing this option. The build checks the effective partition selection, QIO Flash mode and rollback setting, including when reusing a configuration:

```
idf.py --board ESP32_S31_MOSAICO -B "$ESP_VISION_WORK_DIR/build/irisfw" \
  -D ESP_VISION_ENABLE_IRISFW=ON build
```

Retained layout

The 16 MiB layout follows the fixed system prefix of Mosaico Recovery ABI 2, contract `esp-mosaico/v2` and layout ID `mosaico-retained-test-2m-v2`. The partition table remains at `0x8000`. Application NVS keeps its address. The 7 MiB application is followed by one FAT partition using every remaining byte through the 16 MiB Flash boundary; there is no separate `ui_apps` reservation:

The map below includes the areas before the named partitions. The bootloader row is its reserved address window, not the actual image size; neither it nor the partition-table sector is an application partition. The Flash ends at `0x1000000` (exclusive).

Table 3: Complete Iris Flash map

Region	Type/subtype	Offset	Size
Reserved prefix	Outside named partitions	0x0	0x2000 (8 KiB)
Retained bootloader window	Outside named partitions	0x2000	0x6000 (24 KiB)
Partition table	Outside named partitions	0x8000	0x1000 (4 KiB)
otadata	data/ota	0x9000	0x2000 (8 KiB)
phy_init	data/phy	0xB000	0x1000 (4 KiB)
sysmeta	data/nvs	0xC000	0x14000 (80 KiB)
vibe_mode	app/test	0x20000	0x1C0000 (1.75 MiB)
coredump	data/coredump	0x1E0000	0x20000 (128 KiB)
nvs	data/nvs	0x200000	0x10000 (64 KiB)
main_app	app/ota_0	0x210000	0x700000 (7 MiB)
ffat	data/fat	0x910000	0x6F0000 (6.9375 MiB)

The fixed `0x210000` application start leaves only 13.9375 MiB for application and FAT combined. Two 7 MiB partitions would exceed 16 MiB Flash by 64 KiB, so the default uses 7 MiB for the application and 6.9375 MiB for FAT.

The packer rejects applications larger than the 7 MiB slot. This layout reclaims the former UI resource area for FAT. Changing an existing FAT partition's offset or size does not migrate or resize its filesystem; preserve required files before installing a different layout.

Iris builds load Wi-Fi directly from NVS partition `sysmeta`, namespace `shared`, string keys `wifi_ssid` and `wifi_password`. A valid profile is enabled by default. The legacy `sysmeta/wifi` keys `ssid`/`password` are read only when both shared keys are absent; startup does not copy credentials or write NVS. ESP-VISION's ordinary `nvs/ev_network` record and factory credentials are not used in this build. Missing credentials leave Wi-Fi unconfigured; incomplete, oversized or unreadable data produces a configuration error and is left untouched.

ESP-VISION network configuration writes the same shared credential keys. Its `ev_enabled` (u8) and `ev_revision` (u32) keys preserve the service's enable/revision state; Vibe Mode continues to use the shared SSID/password. Forget stores an empty shared pair to prevent either firmware from reviving legacy credentials. No operation erases `sysmeta` or unrelated shared settings.

These packages require an existing compatible Vibe Mode installation, version 0.2.0 or later. Install the `.irisfw` through its native System Update path. This build's raw IDF flashing manifest is not a Vibe Mode installation package: its bootloader and blank OTA metadata must not replace the retained system. Changing from the ordinary factory layout does not migrate existing files. The new `ffat` follows the board's filesystem initialization policy; it is never supplied as an empty image in the update package.

Direct application flashing

The root `app` and `app-flash` commands also support ordinary firmware on all supported boards; see *Application-only Flashing*. This section describes the Iris-specific procedure for a device already using the Iris layout and booting `main_app`. First confirm the board, running application and partition layout; the source of truth is `boards/ESP32_S31_MOSAICO/port/partitions-16MiB-irisfw.csv`. App-only flashing is not a conversion from the ordinary factory layout.

1. Select a supported ESP-IDF environment and set `ESP_VISION_WORK_DIR` to a writable workspace outside the checkout. Keep this Iris build separate from ordinary builds and explicitly enable `ESP_VISION_ENABLE_IRISFW`. The application must come from this Iris configuration; an ordinary factory build or a complete `.irisfw` archive is not an application image for direct writing.
2. Release other clients and serial monitors. For a running ESP-VISION application, the installed CLI's `control --action bootloader` enters ROM download mode; see *Device management CLI*. Wait for USB to re-enumerate, identify the same board and set `PORT` to its download port. Do not assume the application port name stays the same. This operation does not enter Vibe Mode.
3. From the repository root, run the command below. IDF builds the application as needed and flashes only that application.
4. After reset, reconnect through the application's port and verify its board/version and running `main_app` address. For camera changes, also capture a fresh camera frame. A completed write alone does not prove the expected application booted.

```
idf.py --board ESP32_S31_MOSAICO -B "$ESP_VISION_WORK_DIR/build/irisfw" \
  -D ESP_VISION_ENABLE_IRISFW=ON -p "$PORT" app-flash
```

The IDF target obtains the application offset from this build's partition table: `0x210000` for `main_app`. It writes `esp-vision.bin` only, retaining the installed bootloader, partition table, OTA metadata, Vibe Mode, NVS and FAT. With the Iris option enabled, `flash` (including `build flash monitor`) selects the same application-only target and prints this choice. Subsequent invocations recognize the option saved in that build directory's CMake cache; an explicit `-D` value takes precedence. Use separate build directories for ordinary and Iris firmware. Ordinary builds retain the usual full `flash` behavior.

Direct flashing uses the chip's ROM download interface, which is distinct from Vibe Mode. ESP-VISION's USB bootloader control can enter ROM download mode; select the re-enumerated download port before flashing. Direct app flashing neither installs this layout on a factory-layout board nor changes which application is selected to boot. It is suitable for runtime development tests, but does not exercise the native Iris update transaction, HEALTHY acceptance or installation receipt. Validate those separately by manually entering Vibe Mode and installing an `.irisfw` package.

Native Iris installation acceptance

After MicroPython and EV-MUX become ready, the Iris-enabled application accepts its pending OTA image and records its address in the existing ABI 2 recovery metadata. The same macro enables a minimal Iris 0.2 console acceptance subset on the existing USB CDC interface: hardware/firmware identity, authentication binding, BOOT/LINK_READY/HEALTHY events, status, ping/time and read-only system inventory. The native updater can check the new boot, exact ELF identity, partition-table hash and retained operation receipt without a host modification. HEALTHY requires confirmed startup, VM readiness and a non-crash reset; it does not prove peripheral behavior.

The compatibility code shares EV-MUX's first-come ownership lease and TX scheduler. Iris records are parsed only outside EV-MUX frames, and never interleave bytes with a frame already being sent. An Iris owner does not enable EV-MUX REPL, logs or preview; disconnect or lease expiry releases ownership. The Iris build adds an `ESP-Iris` USB product marker for native discovery, retaining the existing CDC interface. Ordinary builds compile none of this adaptation. No Iris runtime, second USB stack, application updater or remote Vibe Mode command is added.

Enter Vibe Mode manually with AI held while pressing Reset before installing a subsequent package. The application rejects Iris restart and update-write commands. After the native updater releases the device, ESP-VISION clients continue to use EV-MUX. Its `firmware.status.v1` capability provides additional read-only evidence, including the exact application BIN hash; see *Frame format, logical channels and RPC protocol*.

Container and publication

The unsigned ZIP uses `esp-iris-system-update/0.2` and contains exactly `manifest.json`, `partition-table.bin` padded to 4096 erased bytes, and `app.bin`. The manifest declares `minimum_recovery_version=0.2.0`. Bootloader, OTA metadata, Vibe Mode, NVS, coredump, FFAT and UI resource images are excluded. The packer checks the retained partition policy and rejects extra flash inputs except the standard blank OTA metadata generated by IDF. Secure Boot signatures and encrypted images need a separate policy and are rejected.

The Iris build retains the standard ESP-IDF application descriptor's project and version. Tags must fit its 31-byte version and match the compiled descriptor exactly. Build provenance binds the compile profile, board, source commit, pipeline/job IDs, actual IDF commit, configuration/layout hashes and all input hashes. It excludes personal paths and configuration contents. Packaging consumes the exact pipeline artifact without rebuilding or selecting another IDF directory.

The format reference is esp-mosaico-utils at be151ae047014e79a9c7ae0092dd203844499003. The packer and independent validator use the standard library plus hash-pinned YAML parsing. `dist/irisfw/` contains one board's package, `.provenance.json` and `.validation.json` sidecars, `SHA256SUMS` and `release.json`.

`publish_irisfw` waits for `qualify_firmware_release` and uses `CI_JOB_TOKEN` to publish the frozen files to the project's Generic Package Registry under package `esp-vision-irisfw`, version `<tag>`. The download endpoint is `${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/generic/esp-vision-irisfw/<tag>/<filename>` and uses existing registry access permissions. Existing Launchpad BIN paths and the external DL uploader are unchanged.

Publication preflights every remote file, reuses identical bytes and rejects conflicting bytes. It compares every downloaded upload before writing `release.json` last. Retry with the same frozen artifacts; a different pipeline or rebuilt provenance requires a new version. A CI resource group serializes publication. Checksums establish integrity, not installation authorization. Platform upload, approval and public listing remain separate from registry publication and device testing.

To package downloaded artifacts from one pipeline:

```
python -m pip install --require-hashes -r tools/ci/irisfw/requirements.lock
python tools/ci/irisfw/package_release.py --tag "$CI_COMMIT_TAG" \
  --source-sha "$CI_COMMIT_SHA" --pipeline-id "$CI_PIPELINE_ID" \
  --build-root build --output dist/irisfw
python tools/ci/irisfw/validate_release.py --tag "$CI_COMMIT_TAG" \
  --source-sha "$CI_COMMIT_SHA" --pipeline-id "$CI_PIPELINE_ID" --root dist/irisfw
```

Use a new output directory per candidate. `ESP_VISION_IRISFW_ONLY=1` with `ESP_VISION_TEST_BOARD=ESP32_S31_MOSAICO` and `ESP_VISION_TEST_IDF_IMAGE=release-v6.1` selects only the Iris firmware build in API/web CI; it does not operate hardware. Offline regressions run with `python -m unittest tools.tests.test_irisfw_release`. Branch/MR pipelines do not publish registry files.

7.6 Introduce a New Model

ESP-VISION supports two model runtime paths: ESP-DL `.espdl` models use the *espdl –Model Inference* module, and TensorFlow Lite `.tflite` models use the *tflite –Model Inference* module. Model files are not built into the firmware; they live on board storage and are loaded at runtime. This guide adds a new model and runs it.

7.6.1 1. Obtain or Convert the Model

Choose the model runtime first:

- ESP-DL: get a ready `.espdl` from the ESP-DL model zoo, or convert your own model to the `.espdl` format with the ESP-DL quantization/export toolchain, matching the selected chip (ESP32-P4, ESP32-S3, or ESP32-S31).
- TFLite Micro: use a TensorFlow Lite `.tflite` flatbuffer that fits TensorFlow Lite Micro and the enabled operator set. Quantized int8 models are usually the practical target on the current boards.

Keep the directory layout under `models/` in the repository when adding shared assets, mirroring `models/espdet/` and `models/tflite/`.

7.6.2 2. Copy the Model to Board Storage

Place the `.espdl` or `.tflite` file on storage the firmware can read, such as `/sdcard` or `/flash`:

- SD card: copy the file onto the card, which mounts at `/sdcard`.
- On-flash FAT (`ffat`): the data partition is exposed over USB MSC, so you can drag the file onto the mass-storage drive; it is visible as `/flash`.

7.6.3 3. Pick the Right API

Choose the API that matches the runtime and task:

Task	API	Result
ESP-DL object detection (ESPDet)	*espdl.ESPDet*	`(x, y, w, h, score, category)`
ESP-DL object detection (YOLO11)	*espdl.YOLO11*	`(x, y, w, h, score, category)`
ESP-DL pose detection	*espdl.YOLO11nPose*	detection plus 17 COCO keypoints
ESP-DL image classification	*espdl.ImageNetCls*	`(label, score)`
ESP-DL with Python decoding	*espdl.Model*	raw output bytes with tensor metadata
Generic TFLite Micro execution	*tflite.Model*	raw output tensors, or a callback result

For ESP-DL wrappers, pass `mean`, `std`, `rgb_swap`, `score`, `nms`, `topk`, or `softmax` to the constructor when the model needs different preprocessing or filtering. When you need to keep ESP-DL inference but decode outputs in Python, inspect `espdl.Model.inputs()` and `espdl.Model.outputs()`, then decode the `RawTensor` bytes returned by `predict()`. For TFLite Micro models, inspect `input_shape`, `input_dtype`,

`input_scale`, `input_zero_point`, `output_shape`, `output_dtype`, `output_scale`, and `output_zero_point` and implement the model-specific preprocessing or post-processing in Python or helper code.

7.6.4 4. Run ESP-DL Inference

```
import sensor, image, espdl

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)

det = espdl.ESPDet("/sdcard/my_model.espdl", score=0.5, nms=0.45)

while True:
    img = sensor.snapshot()
    for x, y, w, h, score, category in det.detect(img):
        img.draw_rectangle(x, y, w, h, color=(255, 0, 0))
    img.flush()
```

7.6.5 5. Decode ESP-DL Outputs in Python

Use *`espdl.Model`* when the model can use ESP-DL image preprocessing and inference, but its output tensors need Python-side decoding.

```
import sensor, espdl

model = espdl.Model("/sdcard/my_model.espdl", mean=(0, 0, 0), std=(255, 255, 255),␣
↪letterbox=True)
try:
    print("inputs:", model.inputs())
    print("outputs:", model.outputs())
    img = sensor.snapshot()
    outputs = model.predict(img)
    for name, tensor in outputs.items():
        _, shape, dtype, exponent, raw = tensor
        print(name, shape, dtype, exponent, len(raw))
finally:
    model.deinit()
```

Each output tensor is returned as raw bytes plus `shape`, `dtype`, and ESP-DL `exponent` metadata. Decode the bytes according to the tensor type, apply the exponent scale, and then run model-specific post-processing such as sigmoid, box decode, NMS, classification top-k, or coordinate unletterboxing.

7.6.6 6. Run TFLite Micro Inference

```
import tflite

def fill_input(buffer, shape, dtype_code):
    # Fill model-specific quantized input bytes.
    ...

model = tflite.Model("/sdcard/my_model.tflite")
try:
    print("input:", model.input_shape, model.input_dtype, model.input_scale, model.
↪input_zero_point)
    print("output:", model.output_shape, model.output_dtype, model.output_scale,␣
↪model.output_zero_point)
```

(continues on next page)

(continued from previous page)

```
    outputs = model.predict([fill_input])
    raw = outputs[0]
finally:
    model.deinit()
```

See `example/03-Machine-Learning/00-ESP-DL/` for ESP-DL scripts (`espdet_pico.py`, `espdet_pico_python.py`, `yolo11.py`, `yolo11n_pose.py`, `imagenet_cls.py`), and `example/03-Machine-Learning/01-TFLite/` for TFLite Micro scripts (`person_detection.py` and `sine.py`).

7.6.7 7. Optional: Profiling and Validation

Use *`espdl.load_model()`* with `profile=True` to emit ESP-DL profiling output when verifying a new `.espdl` model's performance. For TFLite Micro models, print `model.len`, `model.ram`, input metadata, and output metadata to verify flash size, arena size, tensor layout, and quantization before tuning post-processing.

7.7 Train an ESPDet Pico Model

ESP-VISION runs ESPDet Pico object detection models through *`espdl.ESPDet`*. Model training, export, and quantization are handled by the ESP-Detection project. ESP-VISION loads the exported `.espdl` file and runs real-time inference through the `sensor`, `image`, and `espdl` modules.

This guide uses a single-class object detection model as an example and covers the complete workflow from environment setup, dataset preparation, training, quantized export, and running the model on ESP-VISION.

7.7.1 1. Prepare the ESP-Detection Environment

Use the following training repository:

```
git clone -b feat/esp-vision-train https://github.com/YanKE01/esp-detection.git
cd esp-detection
```

ESP-Detection uses a Python training environment. ESP-IDF is not required for training. ESP-IDF is only required when building ESP-VISION firmware or running a C++ example project generated for ESP-DL.

ESP-Detection uses `uv` to manage dependencies. Select the installation command for the operating system.

Linux/macOS:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Windows PowerShell:

```
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

After installation, reopen the terminal or refresh the current terminal's `PATH` as instructed by the installer. After confirming that `uv` is available, synchronize the project environment:

```
uv sync
```

`uv sync` installs Ultralytics, PyTorch, ONNX, ONNX Runtime, OpenCV, and `esp-ppq`. Training and quantization are much faster on machines with an NVIDIA GPU. CPU execution is also supported, but training time is significantly longer.

After the environment is ready, check that the command entry points are available:

```
uv run python espdet_train.py --help
uv run python -m deploy.quantize_aligned --help
```

7.7.2 2. Prepare the Dataset

When preparing a dataset with Roboflow, export it in `YOLOv11` format. After export, organize the dataset in the ESP-Detection repository, for example:

```
datasets/my_object/
  images/
    train/
    val/
  labels/
    train/
    val/
```

ESP-Detection reads the YOLO detection label format. Each image has a same-named `.txt` label file under `labels`. For example, `images/train/0001.jpg` corresponds to `labels/train/0001.txt`. Each line in a label file describes one bounding box:

```
class_id x_center y_center width height
```

`x_center`, `y_center`, `width`, and `height` are floating-point values normalized by image width and height, usually in the `0.0` to `1.0` range.

For example, a single-class safety helmet dataset can use:

```
0 0.5123 0.4381 0.1840 0.1365
```

Dataset labels should match the class names shown later on the ESP-VISION side.

Then create a dataset configuration under `cfg/datasets/`, for example `cfg/datasets/my_object.yaml`:

```
path: datasets/my_object
train: images/train
val: images/val
test:

names:
  0: my_object
```

`path` is the dataset root directory. `train` and `val` are image directories relative to `path`. The order of `names` is important because it defines the meaning of the `category` value returned by `espdl.ESPDet.detect()` on the board. Keep this order stable after training.

For a multi-class model, add more classes:

```
names:
  0: helmet
  1: person
  2: vest
```

7.7.3 3. Train ESPDet Pico

ESP-Detection provides `espdet_train.py` as the train-only entry point. It builds the ESPDet Pico network from `cfg/models/espdet_pico.yaml` and prints the best checkpoint path.

Single-class training example:

```
uv run python espdet_train.py \
  --class_name my_object \
  --pretrained_path None \
  --dataset cfg/datasets/my_object.yaml \
  --size 320 320
```

Key parameters:

Parameter	Description
`--class_name`	Training task name. It affects the Ultralytics output directory name.
`--pretrained_path`	Pretrained checkpoint path. Use `None` when no pretrained checkpoint is available; the script builds a new model from `cfg/models/espdet_pico.yaml`.
`--dataset`	Dataset YAML path.
`--size`	Input size in `height width` format. For example, `320 320` means 320x320 input.

The training script uses long training by default, with a large batch size, RAM cache, and data augmentation. For small datasets, monitor validation curves to avoid overfitting. When training finishes, the terminal prints output similar to:

```
Training complete: runs/detect/my_object
Best weights: runs/detect/my_object/weights/best.pt
```

Use `weights/best.pt` for quantization.

7.7.4 4. Prepare Calibration Data

Exporting `.espdl` requires int8 quantization, and quantization requires calibration images. Calibration images should cover the real deployment distribution, including different lighting conditions, distances, angles, backgrounds, and object sizes.

The simplest approach is to use validation images directly:

```
datasets/my_object/images/val
```

A separate calibration directory can also be prepared:

```
deploy/my_object_calib/
  0001.jpg
  0002.jpg
  0003.jpg
```

Calibration images do not need labels, but their distribution should be close to actual camera input. If the calibration set is too narrow, simulator metrics may look normal while on-board scores are low or detections are missing.

7.7.5 5. Quantize and Export .espdl

Use `deploy.quantize_aligned` to export the trained `best.pt` checkpoint to an ESP-DL `.espdl` model:

```
uv run python -m deploy.quantize_aligned \
  --model runs/detect/my_object/weights/best.pt \
  --size 320 320 \
  --target esp32p4 \
  --calib_data datasets/my_object/images/val \
  --espdl deploy/espdet_pico_320_320_my_object.espdl \
  --data cfg/datasets/my_object.yaml
```

Parameters:

Parameter	Description
`--model`	Trained `best.pt` checkpoint. An exported ONNX model can also be used.
`--size`	Model input size. It must match the training size.
`--target`	Target chip, such as `esp32p4` or `esp32s3`. It should match the chip that will run the model.
`--calib_data`	Calibration image directory.
`--espdl`	Output ESP-DL model path.
`--data`	Dataset YAML. When provided, the script also reports simulator validation metrics.

The quantization script prepares ONNX from `pt` internally, then exports an int8 ESP-DL model through ESP-PPQ. The current configuration uses percentile calibration, bias correction, and fusion settings tuned for scale consistency. The script prints the exported model's operator histogram. When `--data` is provided, it also prints simulator `mAP50` and `mAP50-95`.

After quantization, confirm that the `.espdl` file exists:

```
ls -lh deploy/espdet_pico_320_320_my_object.espdl
```

7.7.6 6. Add Model Metadata

To publish the model as a shared ESP-VISION asset, place it under `models/espdet/pico/<name>/`:

```
models/espdet/pico/my_object/
  README.md
  espdet_pico_320_320_my_object.espdl
  espdet_pico_320_320_my_object.json
```

The `.espdl` file is the model binary. The `.json` file is sidecar metadata read by model lists and tooling. Minimal JSON example:

```
{
    "name": "ESPDet Pico My Object",
    "architecture": "ESPDet Pico",
    "api": "espdl.ESPDet",
    "task": "detection",
    "input": "320x320",
    "inputFormat": "RGB565",
    "dataset": "My Object",
    "labels": ["my_object"],
    "sizeBytes": 574864,
    "description": "Detects my_object in camera images."
}
```

`labels` must follow the same order as `names` in the dataset YAML. `sizeBytes` should be the actual model file size. On Linux, use:

```
stat -c '%s' models/espdet/pico/my_object/espdet_pico_320_320_my_object.espdl
```

`README.md` should describe the model purpose, dataset source, input size, quantization metrics, and basic usage. For public datasets, keep the dataset link in README instead of the JSON file.

7.7.7 7. Copy the Model to Board Storage

ESP-VISION loads models from the file system at runtime. Two common paths are:

Path	Use
`/sdcard`	SD card. Suitable for multiple models, large files, or frequent model replacement.
`/flash`	On-flash FAT data partition. It is usually exposed as a USB MSC drive and is suitable for a small number of frequently used models.

For example, after copying the model to the SD card root directory, use:

```
MODEL = "/sdcard/espdet_pico_320_320_my_object.espdl"
```

If the model is copied to the flash root directory, use:

```
MODEL = "/flash/espdet_pico_320_320_my_object.espdl"
```

7.7.8 8. Run on ESP-VISION

The following is a complete MicroPython inference script:

```
import espdl
import sensor
import time

MODEL = "/sdcard/espdet_pico_320_320_my_object.espdl"
LABELS = ("my_object",)

sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.skip_frames(time=1000)

det = espdl.ESPDet(MODEL, score=0.5, nms=0.7)

try:
    while True:
        img = sensor.snapshot()
        for x, y, w, h, score, category in det.detect(img):
            label = LABELS[category] if category < len(LABELS) else str(category)
            img.draw_rectangle(x, y, w, h, color=(255, 0, 0), thickness=2)
            img.draw_string(x, max(0, y - 12), "%s %.2f" % (label, score),␣
↪color=(255, 0, 0))
        img.flush()
        time.sleep_ms(20)
finally:
    det.deinit()
```

`sensor` output should use `RGB565`. `score` is the confidence threshold, and `nms` is the NMS threshold. During the first on-board test, start with a lower `score` to check whether the model can produce stable boxes, then adjust it based on false positives and missed detections.

7.7.9 9. Validate and Tune

Use the following validation order:

- Check the training log and validation metrics first to confirm that the floating-point model does not obviously underfit or overfit.
- Check the simulator metrics reported by the quantization script to confirm that the int8 model does not drop significantly.

- Run the model with real camera input on ESP-VISION, and inspect detections at different distances, angles, and lighting conditions.
- Tune `score` and `nms`, then freeze the final model, labels, and example script.

If simulator metrics are normal but on-board results are poor, check these items first:

- Whether `--size` exactly matches the training size.
- Whether `labels` follows the same order as `names` in the dataset YAML.
- Whether calibration images cover the real camera input distribution.
- Whether the board camera image has exposure issues, color issues, or objects that are too small.
- Whether `score` is too high and filters out low-score detections.

After the model file, JSON metadata, README, and test script are verified, the model can be added to ESP-VISION's `models/` and `example/` directories as a reusable model.

7.8 Add a New Board

A board package lives in a single tree, `boards/<BOARD>/`: the ESP-VISION files at the top level and the MicroPython ESP32 port files under the `boards/<BOARD>/port/` subdirectory. The build projects `boards/<BOARD>/port/` onto `ports/esp32/boards/<BOARD>/` of the generated MicroPython copy. Start from the `TEMPLATE` board.

7.8.1 MicroPython Port Side

In `boards/<BOARD>/port/`:

File	Purpose
`mpconfigboard.cmake`	`IDF_TARGET` value and `SDKCONFIG_DEFAULTS` chain.
`mpconfigboard.h`	MicroPython feature flags and USB strings.
`sdkconfig.board`	Board-specific ESP-IDF Kconfig overrides.
`partitions-*.csv`	Partition table.
`board.json`, `board.md`	Upstream board manifest metadata.

7.8.2 ESP-VISION Side

In `boards/<BOARD>/`:

File	Purpose
`boardconfig.h`	Pin assignments and board runtime constants.
`imlib_config.h`	OpenMV `imlib` feature switches.
`manifest.py`	Frozen Python modules.
`camera.c`	Board-specific camera backend.
`display.c`	LCD panel and backlight implementation.
`sdcard.c`	SD card power and card-detect implementation.

`micropython.cmake` automatically picks up `camera.c`, `display.c`, and `sdcard.c` from the board directory when present, and includes the board's optional `board.cmake`.

7.8.3 Build and Flash

```
idf.py --board <NEW_BOARD> -p /dev/ttyACM0 build flash monitor
```

Note: This page is a starting outline. Detailed bring-up steps (sensor selection, PPA configuration, display timing) will be expanded.

7.9 EV-MUX v4 Host Integration Entry

The complete EV-MUX reference has moved to *Communication System and EV-MUX*. This page preserves existing links; use the new chapter for architecture, interfaces, sessions, protocol and acceptance.

7.10 Agent Code Generation Guide

This guide is normative for AI agents that generate ESP-VISION application code. Treat the target as an ESP32 microcontroller running the ESP-VISION build of MicroPython v1.28.0, not as a PC running CPython. Use only APIs present in this knowledge pack, enabled for the selected board, or demonstrated by a repository example; never assume that a desktop standard-library module, `pip` package, native wheel, operating-system service, subprocess, or PC filesystem is available on the device.

7.10.1 Dependency Rules

The default ESP-VISION firmware cannot dynamically install dependencies from `requirements.txt` and does not include device-side `mip`. Generated target code must not invoke `pip`, parse `requirements.txt`, call `mip.install()`, or silently depend on an unverified third-party package. If a required module is not in the knowledge pack, either rewrite the solution with supported modules or explicitly state that it must be copied to `/lib` with a host tool, precompiled as compatible `.mpy`, frozen into custom firmware, or implemented as a native module before the script can run.

Host-side tooling is separate from target code. A PC utility may use CPython packages such as `pyserial`, but label that code as host-only and never import those packages from the ESP-VISION script.

7.10.2 All-in-One Output

Unless the user asks for a package or a multi-file architecture, return one self-contained `main.py` for the device. Put supported imports, constants, initialization, application logic, the main loop or bounded test, error handling, and cleanup in that file. Avoid helper modules that the user would have to create separately. Models and other binary assets may remain separate files, but list their exact device paths and deployment prerequisites.

Prefer bounded, observable verification code before an infinite product loop. Print a unique success marker only after the camera, model, storage, network, or other requested operation has really succeeded. The following target-side smoke test is an all-in-one MicroPython script and prints `EVTEST:OK` only after capturing a frame:

```
import gc
import sensor


def main():
```

(continues on next page)

(continued from previous page)

```
    print("EVTEST:START")
    sensor.reset()
    try:
        sensor.set_pixformat(sensor.RGB565)
        sensor.set_framesize(sensor.QQVGA)
        sensor.skip_frames(time=500)
        image = sensor.snapshot()
        width = image.width()
        height = image.height()
    finally:
        sensor.shutdown()
    print("EVTEST:OK {}x{} heap={}".format(
        width, height, gc.mem_free()))


main()
```

7.10.3 Device Communication and Acceptance

Agent is a host application alongside IDE and CLI; use Device Core for device interaction rather than another EV-MUX client. The shared host/Core/protocol/firmware relationship is in *Solution Architecture*. Use *Communication System and EV-MUX* for acquisition, upload, execution and application evidence. The complete host-only camera closed-loop example is in *Communication acceptance, closed-loop tests and troubleshooting*. Handshake, upload acknowledgement, echo and REPL prompts do not prove application success.

Chapter 8

Solution Architecture

ESP-VISION separates host tools, online knowledge and device firmware. IDE and custom SDK applications use Device Core directly. AI clients and independent agents use the plugin Skill, its bundled Device CLI for local device operations, and the official online Knowledge MCP over HTTPS. Device CLI calls Device Core; the SDK implements the host side of EV-MUX v4 over board-supported USB CDC, USB Serial/JTAG, UART or TCP. Device firmware combines MicroPython, vision/inference modules, platform services and board backends. Firmware code is further layered by whether it touches MicroPython (`mp_obj_t` / `py/*.h`).

8.1 Layered Overview

- **Host applications**: IDE presents editing/debugging UI; custom applications use the SDK; AI clients and independent agents select tools for the task. Agents own their AI loop.
- **Plugin** (`host/esp-vision-plugin`, `@esp-vision/plugin`): packages one capability-oriented Skill, a fixed Device CLI and the official Knowledge HTTP endpoint configuration for Codex, Claude Code and WorkBuddy, with separate WorkBuddy Expert, QoderWork and Cursor export adapters. Host loading and marketplace approval require platform acceptance.
- **Device CLI** (`host/device-cli`, `@esp-vision/device-cli`): exposes device operations through shell commands, JSON results, streamed events and binary output files. Both the plugin copy and the standalone command use public Device Core APIs.
- **Knowledge MCP** (`host/knowledge-mcp`, `@esp-vision/knowledge-mcp`): provides API, documentation and example queries from a versioned snapshot. The official plugin connects to the online HTTP service; the npm package supports stdio and HTTP for self-hosting. Main builds the snapshot, npm archive and immutable image; website deploys that image. HTTP records anonymous daily aggregates.
- **Device Core** (`host/device-core`): the independently versioned `@esp-vision/device-core` npm package owns protocol parsing, request correlation, sessions, keepalive and operation lifecycles. The diagram selects representative API names instead of listing every method; complete contracts are in *Application API reference*.
- **EV-MUX v4**: the runtime wire protocol between Core and firmware, covering framing, logical channels, ownership and VM epochs across USB CDC, USJ, UART and TCP. Host adapters and network bridges carry bytes without creating another session owner.
- **Device application**: MicroPython scripts run on the board and call its Python bindings. They can continue running after a host disconnects.
- **Bindings** (`modules/`): the `USER_C_MODULES` layer. The main modules `image`, `sensor`, `display`, `espdl`, and `tflite` self-register via `MP_REGISTER_MODULE`. `py_imageio.c` provides the `image.ImageIO` type, and `py_helper.c` is shared helper code. Bindings only do object conversion and light API adaptation; heavy logic lives in pure C or `platform/`.
- **Platform services** (`platform/`): self-written ESP32 glue. `ev_channel.c` / `ev_mux.c` / `ev_control_transport.c` / `ev_stdio.c` (EV-MUX / EV-ATP transport), `preview.c` (EV-MUX JPEG preview), `display.c` (generic display layer), `sdcard.c` (mount at `/sdcard`), `usb_msc.c` (exposes the `ffat` partition over TinyUSB MSC), `usb_auto_download.c` (S3/P4 OTG CDC re-

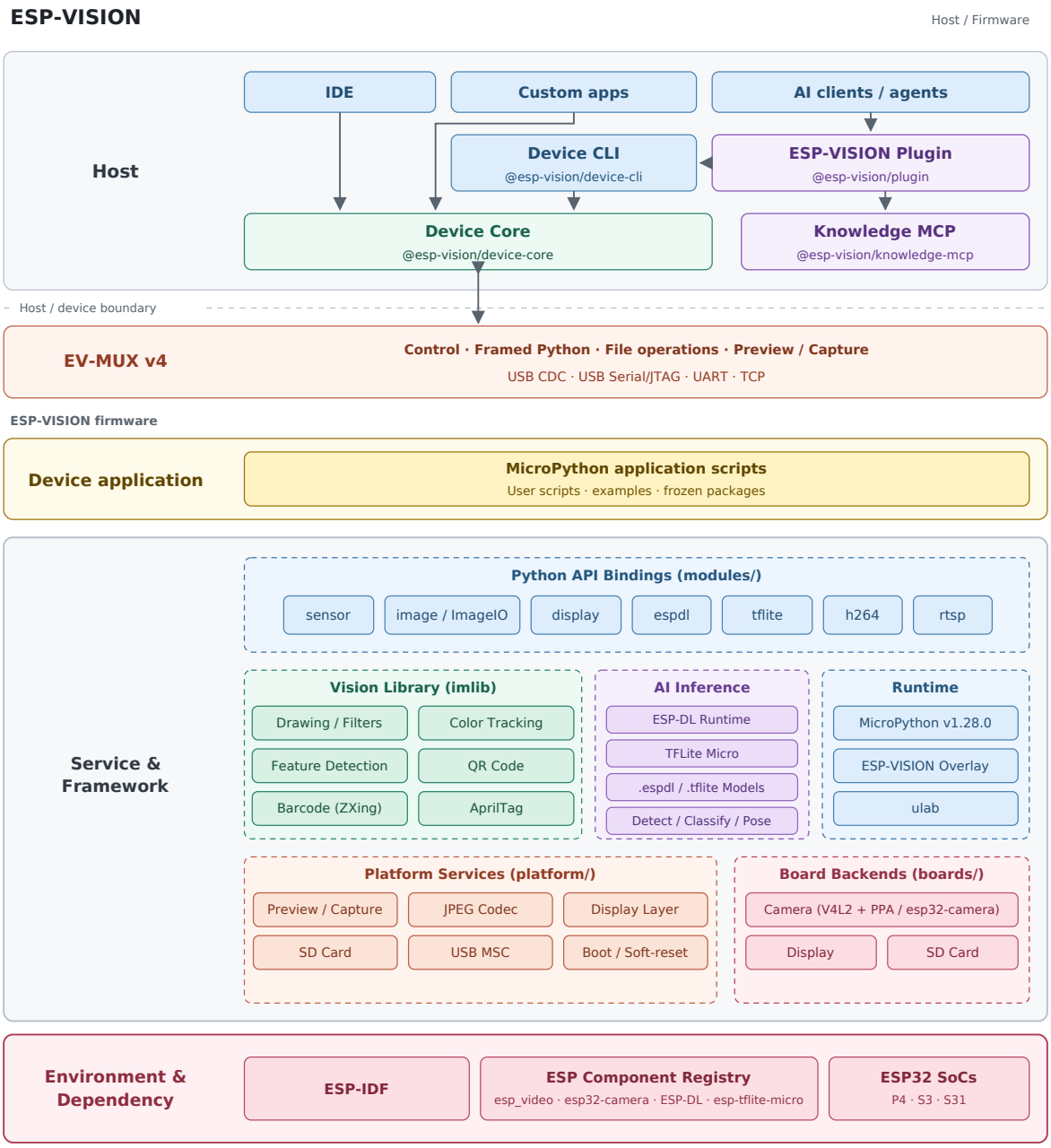


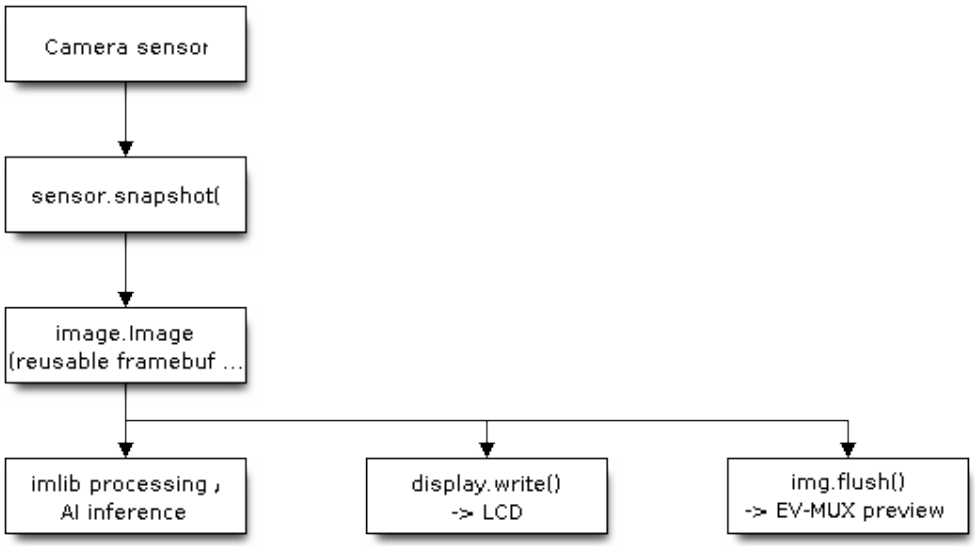
Fig. 1: Local device control through Device CLI / Device Core, with a separate online Knowledge MCP service

set/download protocol), `jpeg.c` (hardware or software JPEG), `debug.c`, and `main.c` (startup init plus the soft-reset loop).

- **imlib component** (`components/imlib/`): pure-C vision algorithms, an IDF component maintained as MIT, derived from OpenMV `lib/imlib`.
- **Board backends** (`boards/<BOARD>/`): per-board configuration and the real camera/display/sdcard implementations. P4X and S31 use `esp_video`/V4L2; P4X also uses PPA, while S3 uses `esp32-camera`.
- **MicroPython + overlay**: MicroPython v1.28.0 is the fixed baseline; project changes live in `overlay/micropython/` and are applied to a generated build copy under `build/micropython/`.

Host-provided byte transports implement physical I/O beneath the SDK protocol. LAN DNS-SD discovery is an enumeration path before connection; ROM firmware flashing uses a separate loader lifecycle and does not send application images through EV-MUX. Firmware and SDK versions remain independent. See *Device Core host SDK* for host integration and API documentation.

8.2 Capture-to-Output Data Flow



`sensor.snapshot()` captures a frame into a reusable framebuffer wrapped as an `image.Image`. Scripts then run `imlib` processing, ESP-DL inference, or TFLite Micro inference on the image, and send it to the LCD with `display.write()` or to the host preview with `img.flush()`.

8.3 Communication Subsystem

EV-MUX v4 owns user/debug streams together and multiplexes channels onto the current USJ, USB, UART or TCP owner link. MicroPython HAL, C stdio, IDF logs and JPEG preview have distinct producer paths.

Detailed architecture, interfaces, acquisition, framing, Python/files/images and acceptance are in *Communication System and EV-MUX*. The current Python traceback/sys.stderr path is documented in *Architecture, Execution Contexts and Standard I/O*. This keeps one canonical communication chapter rather than duplicating it in architecture and Agent guides.

8.4 Source Tree

Path	Responsibility
`host/device-core/`	Public host SDK and transport adapters (`@esp-vision/device-core`).
`host/device-cli/`	Standalone device command toolkit (`@esp-vision/device-cli`).
`host/knowledge-mcp/`	Knowledge MCP runtime, packaged snapshot and server image (`@esp-vision/knowledge-mcp`).
`host/esp-vision-plugin/`	Skill, bundled CLI, Knowledge endpoint configuration and installer (`@esp-vision/plugin`).
`idf_ext.py`	Board-aware `idf.py` extension for the repository root.
`micropython.cmake`	Integration hub: registers user modules, platform and board sources, include paths, conditional `zxing`, and `ulab`.
`lib/`	Pinned third-party submodules (MicroPython, `ulab`, ZXing-C++).
`overlay/micropython/`	ESP-VISION MicroPython delta, using the MicroPython path layout.
`boards/`	Per-board config, frozen manifests, and board peripheral backends.
`platform/`	Shared runtime services (camera, preview, storage, display, USB, JPEG).
`modules/`	MicroPython C/C++ bindings (`sensor`, `image`, `display`, `imageio`, `espdl`, `tflite`, plus chip-dependent `h264` and `rtsp`).
`components/`	ESP-IDF components, including OpenMV `imlib` and the ZXing backend.
`models/`	Optional model assets loaded from board storage at runtime.
`example/`	MicroPython example scripts.
`stubs/`	`.pyi` type stubs describing the C modules.

8.5 Board Composition

A board is defined in a single tree, `boards/<BOARD>/`:

- ESP-VISION side (top level): `boardconfig.h`, `imlib_config.h`, `manifest.py`, and optional `camera.c` / `display.c` / `sdcard.c`.
- MicroPython port side (`boards/<BOARD>/port/`): `IDF_TARGET` value, sdkconfig, partitions, and USB strings. The build projects this subdirectory onto `ports/esp32/boards/<BOARD>/` of the generated MicroPython copy.

See *Add a New Board* for the step-by-step procedure.

8.6 Chip-Dependent Sources

`micropython.cmake` selects modules from `IDF_TARGET` and the board profile. The ESP32-P4 build includes `h264` and `rtsp`; the current P4 board profiles also enable the ZXing-C++ barcode backend. See *Chip and Board Support* for the resulting public API matrix.

8.7 MicroPython Overlay

ESP-VISION uses MicroPython v1.28.0 as a fixed upstream baseline. Project changes to the ESP32 port live under `overlay/micropython/`. The `prepare-micropython` build step applies that tree to a generated copy under `build/micropython/idf<ESP_IDF_VERSION>/micropython/`; the `lib/micropython` submodule remains a clean upstream reference.

For how ESP-VISION relates to its upstream projects, see *Project Relationships*; for the license of each component, see *Licensing*.

The same Device Core firmware API supports two explicit ROM operations: default application-only update with layout matching, and complete installation for blank devices using bootloader/partition/application manifests. Neither operation uses EV-MUX for firmware payloads; IDE/Agent/CLI share its validation and readback logic rather than implementing separate flashing workflows.

Chapter 9

Communication System and EV-MUX

This chapter follows the complete ESP-VISION host communication path: MicroPython I/O, ESP-IDF logs and camera data, physical interfaces, session ownership and framing, through the shared Device Core SDK to IDE, Agent, CLI or custom host applications. It describes the current EV-MUX v4 implementation; firmware release, session version and wire format are separate concepts.

Important: `evMuxVersion:4` defines session/authorization behavior; `EVMUX/1` defines byte framing. v4 still uses EVMUX/1: do not change the header to EVMUX/4. IDE 1.3.0 Device Core accepts only explicitly v4-compatible firmware.

9.1 The complete path at a glance

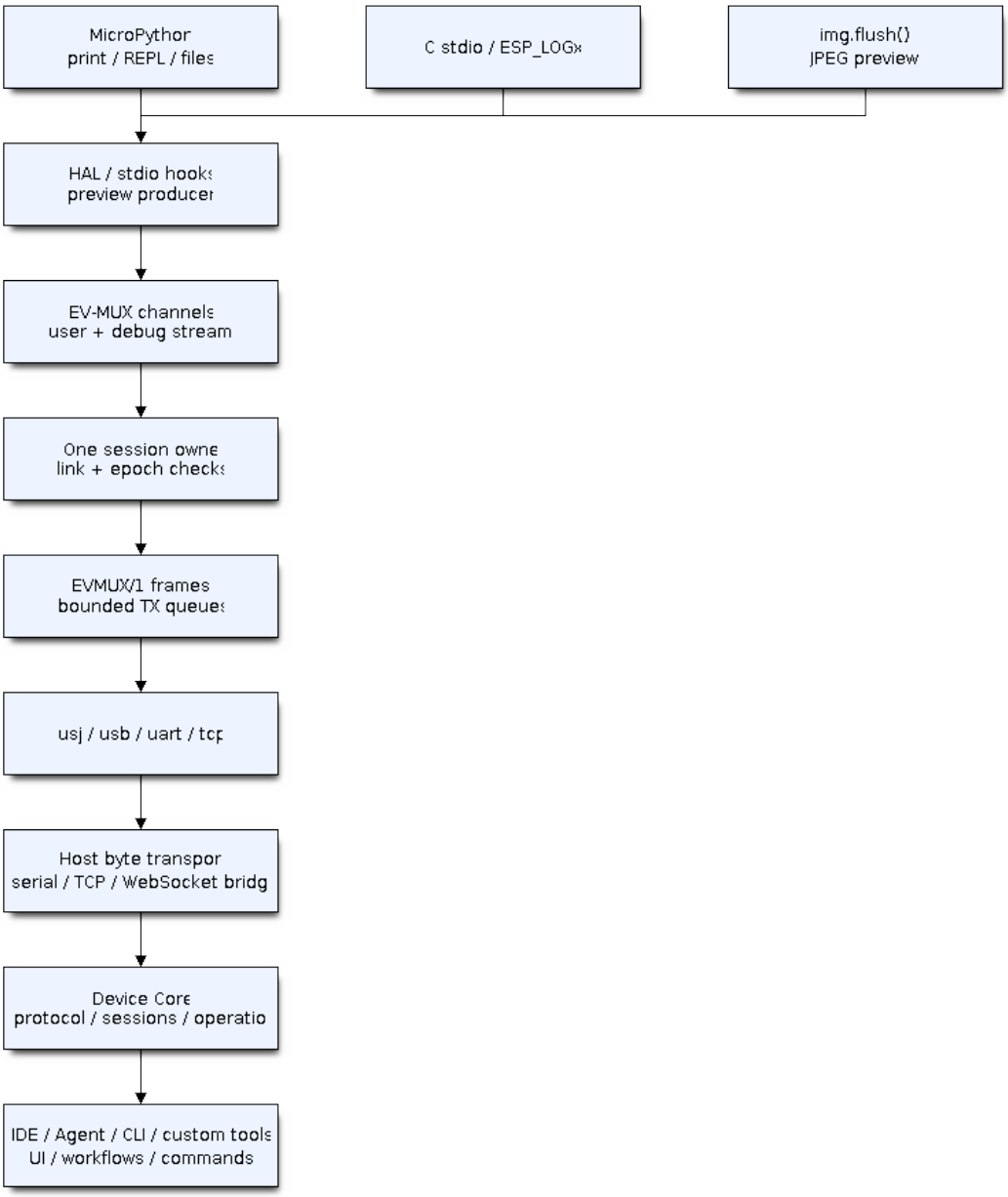


Figure 1: Output is classified, ownership-scoped and framed before reaching one physical connection. Host commands travel in reverse through parser, authorization and REPL/control dispatch. Keep reading because responses, text and images can interleave.

9.2 Recommended reading order

9.2.1 Architecture, Execution Contexts and Standard I/O

System architecture and responsibilities

IDE, Agent, CLI and custom tools are peer host applications using the same Device Core package. Core implements host APIs and protocol/session state; EV-MUX is the runtime wire protocol across USB CDC, USJ, UART and TCP; the firmware implements device operations and the MicroPython runtime. The representative API names shown below are an overview, not a complete inventory.

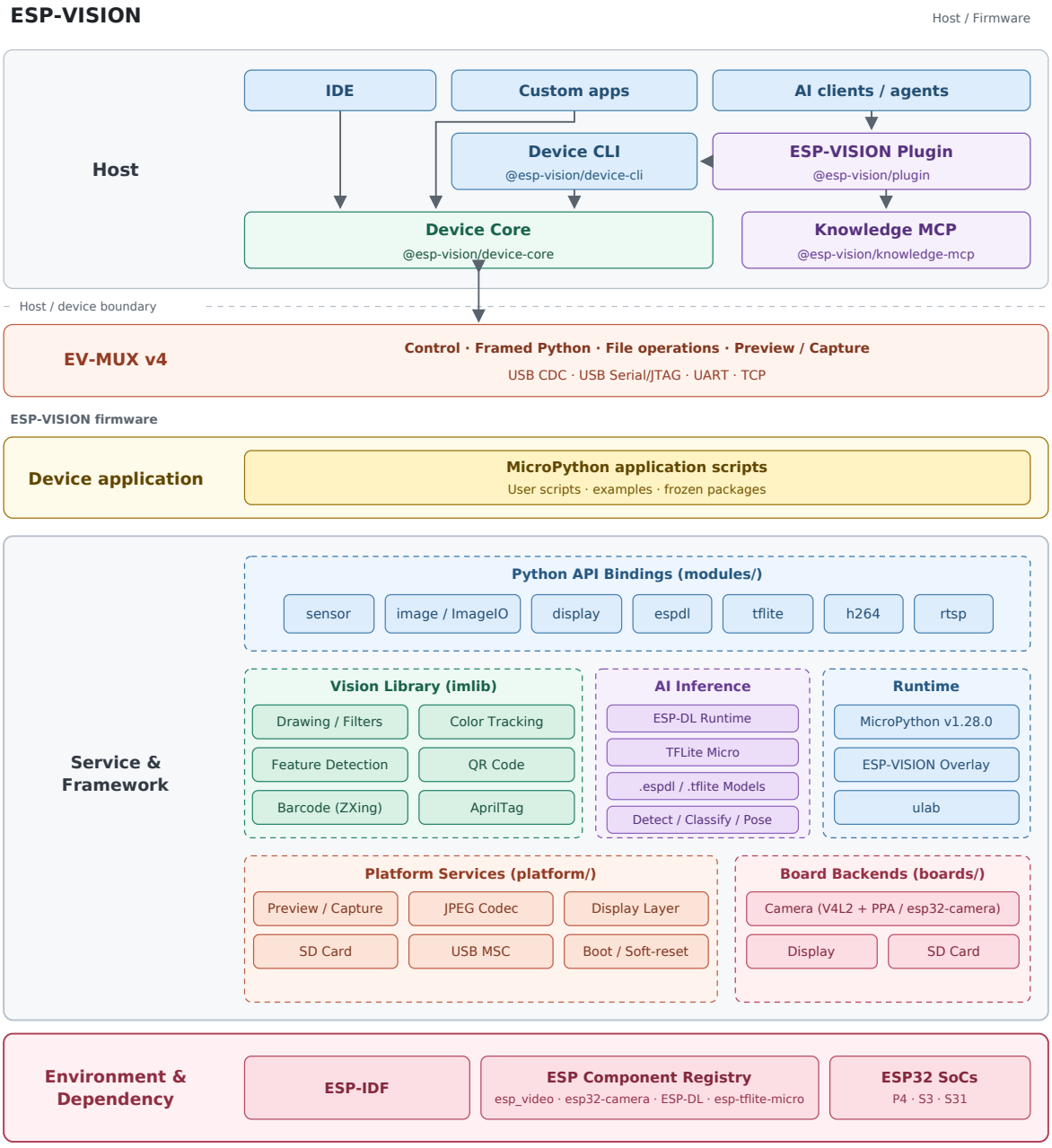


Fig. 1: Shared system diagram; detailed firmware execution paths follow below.

This SVG is shared with the README, system architecture and SDK guides. Discovery and ROM flashing are separate from the EV-MUX runtime path; see *Solution Architecture* for those boundaries. Device scripts execute on the board and do not require a host to remain connected.

Where EV-MUX sits

EV-MUX is runtime byte framing and multiplexing. It neither executes Python nor replaces the existing VM. MicroPython retains parsing, compilation, execution, friendly REPL, stdin/stdout and exception printing; ESP-VISION attaches framing and ownership routing at HAL/platform boundaries.

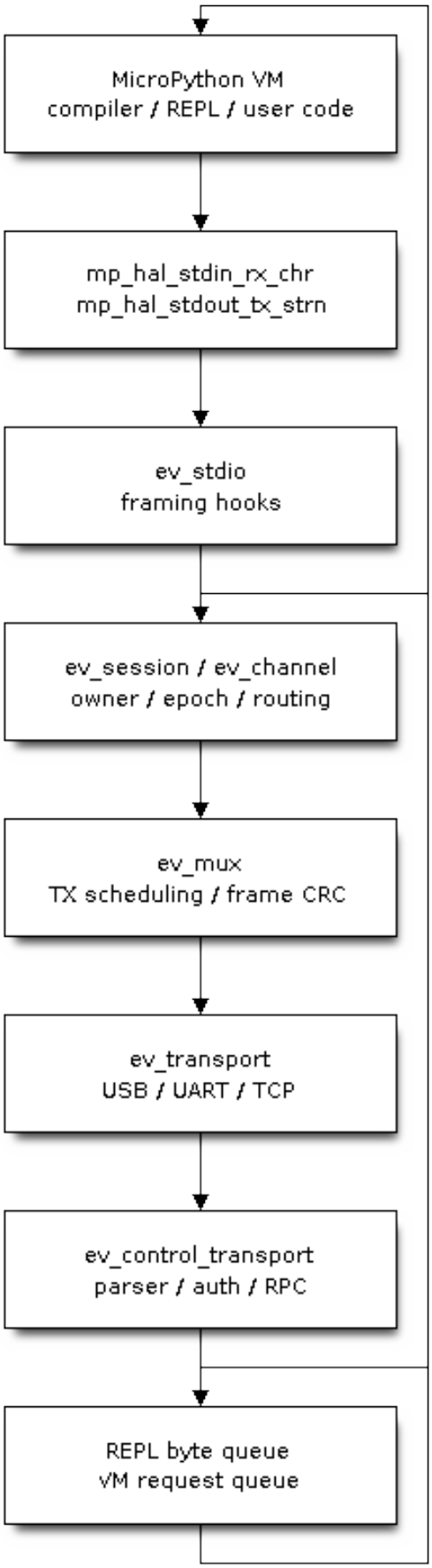

MicroPython VM
compiler / REPL / user code
mp_hal_stdin_rx_chr
mp_hal_stdout_tx_strn
ev_stdio
framing hooks
ev_session / ev_channel
owner / epoch / routing
ev_mux
TX scheduling / frame CRC
ev_transport
USB / UART / TCP
ev_control_transport
parser / auth / RPC
REPL byte queue
VM request queue

Figure 2: Interpreter, HAL, control, ownership/routing and physical backends. stdin/stdout still use the VM; pure transport RPC does not wait for user Python to return.

Output producer, not text content, determines the channel

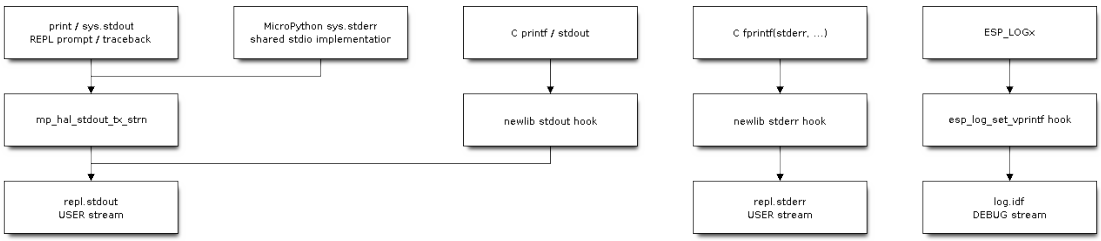


Figure 3: Actual output destinations in this build. Python `print()` and C `printf()` may share a channel but have different call paths. An "error" word does not automatically redirect text to `repl.stderr`.

Table 1: Output path reference

Producer	Firmware entry point	Wire output
Python `print()`, `sys.stdout.write()` and friendly REPL prompts	MicroPython print/stdio → `mp_hal_stdout_tx_strn()` → `ev_stdio_write_repl_stdout()`	`repl.stdout`, `encoding:text`.
Unhandled REPL exceptions and `KeyboardInterrupt` tracebacks	`pyexec.c` uses `mp_obj_print_exception(&mp_plat...)` and HAL stdout.	Can appear on `repl.stdout`; stderr alone is not a failure oracle.
Current MicroPython `sys.stderr.write()`	`shared/runtime/sys_stdio_mphal.c` shares `stdio_write()` for stdout/stderr; both use cooked HAL stdout.	Still `repl.stdout`, rather than desktop CPython independent stderr.
C `printf()` / `fprintf(stdout, ...)`	`CONFIG_LIBC_NEWLIB` installs the newlib `stdout->_write` hook.	`repl.stdout` with `method:c-stdio` for C stdio provenance.
C `fprintf(stderr, ...)`	newlib `stderr->_write` hook.	`repl.stderr`, `method:c-stdio`.
Normal ESP-IDF `ESP_LOGI/W/E` logs	`esp_log_set_vprintf()` → `ev_stdio_log_vprintf()`.	`log.idf`, `method:esp-idf`, debug stream.
Platform logs such as `esp_vision_debug_`	Platform logging calls the EV-MUX IDF-log writer.	`log.idf`; method depends on the producer entry point.
`img.flush()`	Preview producer → JPEG encoder → lossy EV-MUX TX.	`preview.frame` binary JPEG, bypassing printf.

`encoding:text` does not imply UTF-8 character or line boundaries at frame ends. Keep an incremental decoder and unfinished-line buffer rather than decoding each payload independently and accepting replacement characters.

Note: This table describes current source. Channel names do not imply language-level stderr behavior. A future dedicated Python stderr HAL path must update the table, clients and protocol tests.

IDF console versus MicroPython console

IDF system-console Kconfig selects low-level system destinations. MicroPython port settings enable UART REPL, USJ and TinyUSB CDC. EV-MUX ownership selects the runtime business link. These are three separate layers, not aliases of a single switch.

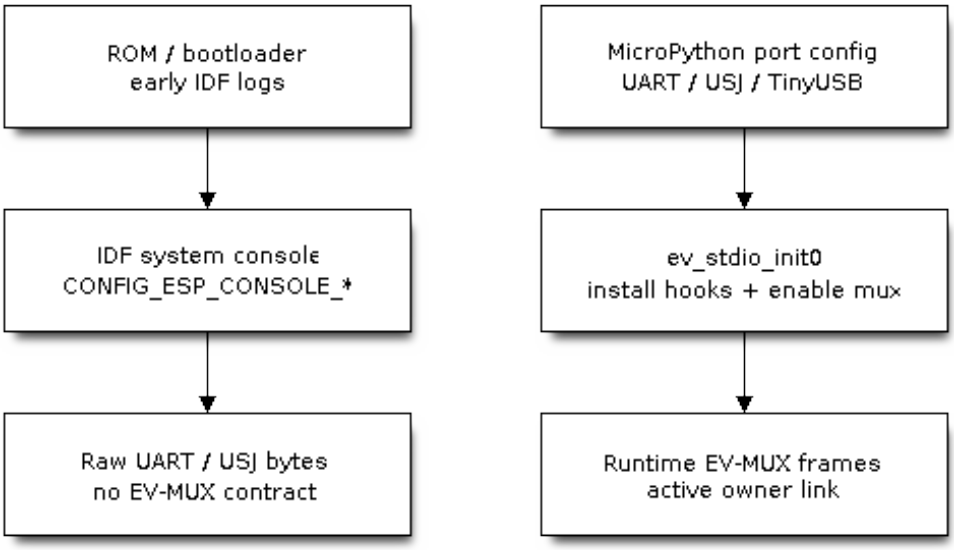


Figure 4: Early system console and runtime EV-MUX have different contracts. Power-on, ROM and panic paths can emit unframed text; runtime business confirmation still requires valid frames.

`MICROPY_HW_ESP_USB_SERIAL_JTAG` controls the ESP-VISION/MicroPython USJ path. Some target/version defaults may still send IDF console output to USJ when it is disabled. Enabled MicroPython USJ initialization explicitly enables peripheral clocks/internal PHY instead of relying on console initialization.

`sensor.evmux(False)` is a runtime diagnostic switch restoring non-framed text paths. `ESP_VISION_EV_MUX_DEFAULT_ENABLED` is local to `ev_stdio.c` and is not currently connected to `boardconfig.h`; defining it there does not disable power-on mux. Disabling framing is not v3 fallback negotiation.

Execution responsibilities

Table 2: Execution contexts

Context	Work	Relationship to user Python
transport task	Read links, frame/authorize ingress, handle discovery, lease/release, network transactions and pure-C transport stats.	These controls remain serviceable during user execution; there is no second VM.
VM task	Friendly REPL, user code and VM RPC touching VFS/objects/camera.	A long script can occupy this task. Queues are bounded; stop it before work requiring an available VM.
TX task	Deliver frames, handle partial writes/backpressure/priority and link loss.	Producer data is copied or has a controlled lifetime, not a dangling VM temporary pointer.
Network worker / TCP manager	Wi-Fi setup, provisioning transactions, connectivity/listener maintenance and slot reclamation.	Shares underlying Python WLAN; ordinary soft reset does not repeatedly initialize the driver.
USB CDC callbacks / restart timer	Record DTR/RTS and defer download/reset work.	Do not tear down USB in its control callback; this is separate from EV-MUX acquire.

`file.write` runs in the transport task and writes native VFS under session/VM generation and storage drain checks; it does not enter the interpreter queue. Legacy `script.write`, `script.run`, `debug.capture_frame`, `debug.capture_screen` and object-related `debug.info` enter the VM queue; `scope:transport.stats` is the exception. Two ring-array slots provide one usable pending request. Queue expiry is 5000 ms; busy/expired work reports `VM_BUSY`/`VM_TIMEOUT`.

LCD screenshots read the display layer's last submitted RGB565 frame and encode JPEG in the same VM task. They do not invoke `sensor` or the camera, change displayed content, subscribe to preview or read back a physical panel. Execution is serialized with display/soft-reset cleanup to avoid reading freed frame buffers.

`repl.signal` Ctrl-C schedules `KeyboardInterrupt` when interpreter interrupts are armed; otherwise it enters REPL editing input. It is not FreeRTOS task preemption and cannot guarantee interruption of a non-returning native call.

Soft-reset and background communication lifetime

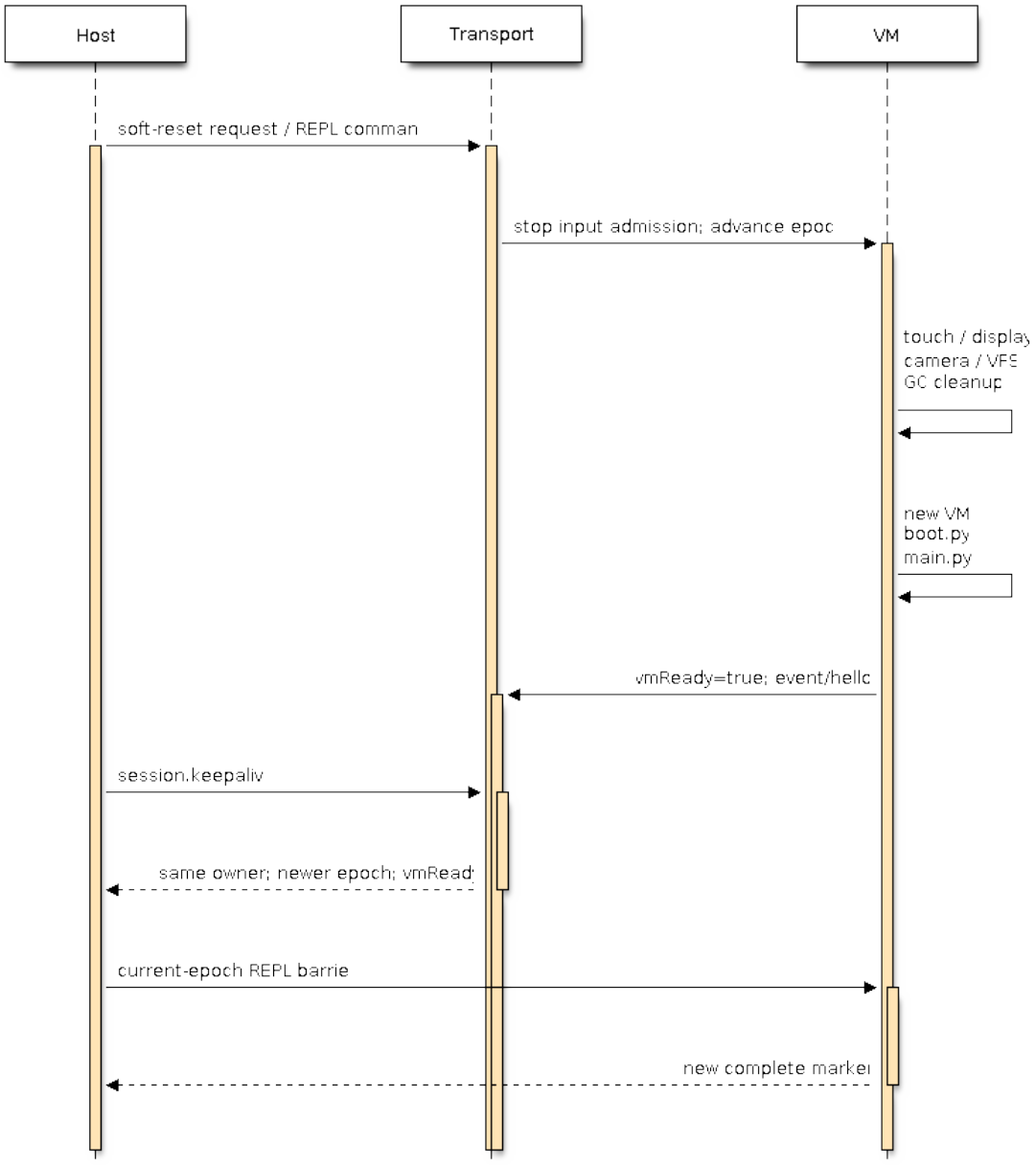


Figure 5: Soft reset advances VM generation while links/network/session can survive. Stop old admission before touch/display/preview/camera cleanup. Epoch advances at reset start, not VM readiness; wait for readiness and real REPL evidence.

Queues, logging failures and absence of raw fallback

Normal queues allow 16 KiB/8 frames per serial endpoint, with two TCP slots sharing a budget. Control reserves another 4 KiB/2 frames. REPL staging uses 512 bytes and an approximately 5 ms coalescing window; preview keeps the latest pending frame: see *Preview, debug capture, backpressure and statistics*.

While mux is enabled, submission/TX failure does not rebroadcast the same Python/C text through raw serial. Hooks guard recursion; error paths use counters instead of more traffic. IDF vprintf has a 512-byte temporary buffer and rejects oversized formatted output with a drop/error, rather than claiming complete delivery.

Early console, mux-disabled paths and low-level panic/ROM are different lifecycle boundaries, not send-failure fallback.

Source entry points

`platform/ev_stdio.c` and `overlay/micropython/ports/esp32/mphalport.c` adapt I/O; `ev_control_transport.c` dispatches ingress; `ev_session.c`/`ev_channel.c` manage ownership/routes; `ev_mux.c` manages framing/snapshots/queues/TX; `ev_vm_output.c` manages VM output scope; `main.c` manages startup/reset. Python stderr is implemented in `lib/micropython/shared/runtime/sys_stdio_mphal.c`.

9.2.2 Physical interfaces, connection generations and build configuration

Warning: TCP control is a wireless development and debug interface for isolated, trusted local networks. It provides no peer authentication or transport encryption. Any reachable client can acquire an available session and perform development operations, including file uploads and script execution. Session ownership and CRC checks do not authenticate the operator. On boards with TCP transport enabled, the listener opens automatically when the enabled background network service is online; a station IPv4 bind does not enforce a same-subnet restriction. Do not expose it to the Internet or untrusted networks. To disable it, set `ESP_VISION_ENABLE_TCP_TRANSPORT` to `0` in the selected board's `boardconfig.h` and rebuild. See the repository's Security Policy for intended use, bounty scope, and private reporting.

Sink types and actual links

`capabilities.sinks` describes compiled backends and their current state. Sink IDs are `usj`, `usb`, `uart` and `tcp`, not OS paths such as `/dev/ttyACM0` or channels. `usb_otg_cdc` is a hardware kind; its endpoint ID remains `usb`. Neither `cdc` nor `usb_otg_cdc` is a substitute.

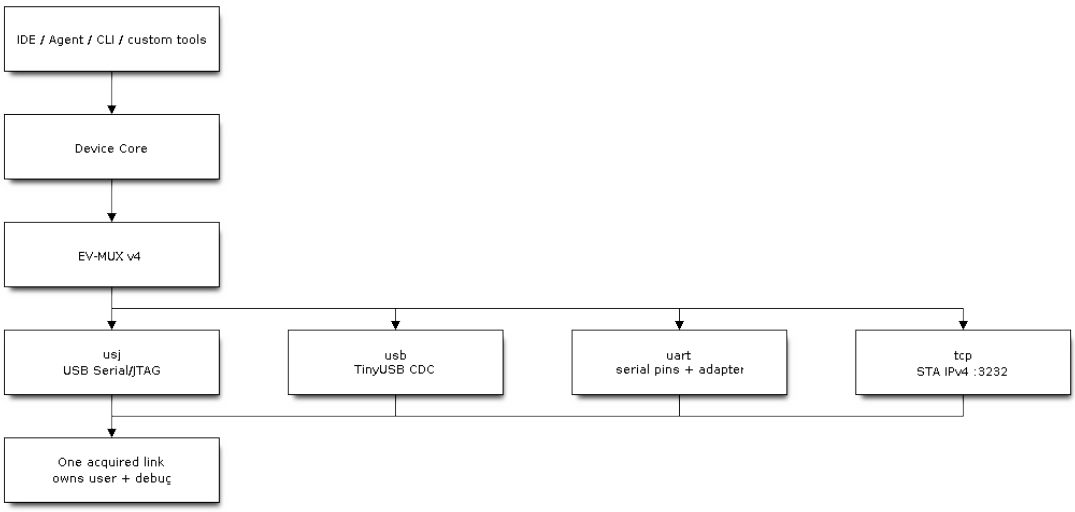


Figure 6: Multiple physical ingress interfaces can discover/acquire the device, which has one control owner. These interfaces do not represent separate VMs.

The host/Core/protocol relationship follows *Solution Architecture*. This figure expands only the physical runtime links beneath EV-MUX; discovery and the ROM loader have separate lifecycles. Serial/socket adapters do not acquire another session.

Table 3: Physical interface comparison

sink	Physical connection	Presence/readiness detection	Typical uses and limitations
`usj`	Chip USB-Serial/JTAG peripheral, usually exposed as a host serial port.	Without reliable unplug detection, `present` is unknown; silent disappearance relies on lease recovery.	P4-EYE default/fallback port; early ROM/IDF output may also appear. It uses a different driver from USB-OTG CDC.
`usb`	USB-OTG uses existing TinyUSB CDC and may share a composite device with MSC.	USB configuration establishes present/ready; actual disconnection invalidates link generation.	Suitable for interaction, files and images; application traffic does not require DTR. CDC and MSC have separate functions/write permissions.
`uart`	Physical UART TX/RX/GND and a USB-UART adapter.	Depends on board UART REPL enablement and initialization; peer disappearance is usually not reliably detectable.	Carries the same channels, subject to baud rate, pin/port allocation and bounded transmission deadlines.
`tcp`	STA IPv4 TCP server on a configured device.	Accept allocates a slot; FIN/RST, socket errors and timeouts reclaim links.	Default port 3232; three accepted slots support up to two read-only preview observers and one controller. Browsers require a WebSocket-to-TCP byte bridge.

Board backend enablement

See *USB Interfaces, Configuration and Disk Names* for USB class composition, configuration, volume labels and fixed MSC LUNs.

Backends depend on `boards/<BOARD>/boardconfig.h`, `boards/<BOARD>/port/mpconfigboard.h` and SoC capabilities. Disabling a MicroPython interface macro disables that interface, rather than only excluding it from EV-MUX; there is no mux-specific serial allowlist. The table describes current source defaults; runtime `capabilities.sinks` is authoritative.

Table 4: Build conditions and current board examples

sink	Build condition	Current board examples and runtime limitations
`usj`	SoC provides USJ and `MICROPY_HW_ESP_USB_SERIAL_JT` is enabled.	P4 boards and `ESP32_S31_CHATBOT`; some raw IDF system-console output remains independently configured.
`usb`	SoC/port provides USB device support and `MICROPY_HW_USB_CDC` is effective.	`ESP32_S3_EYE`, `ESP32_S31_KORVO`, `ESP32_S31_MOSAICO`, `AtomS3R-M12`; P4 boards may expose both CDC and USJ according to SoC support.
`uart`	`MICROPY_HW_ENABLE_UART_REPL` enabled, using board pins/baud rate.	Enabled on P4, `ESP32_S3_EYE`, `AtomS3R-M12`, `ESP32_S31_KORVO`; currently disabled on `ESP32_S31_MOSAICO` and `ESP32_S31_CHATBOT`.
`tcp`	`ESP_VISION_ENABLE_TCP_TRA` with an available WLAN driver.	Listener requires STA IPv4. Current `ESP32_P4X_VISION` port configuration disables WLAN; compiling TCP does not prove runtime network availability.

Examples are `boards/ESP32_P4X_EYE/boardconfig.h`, `boards/ESP32_S3_EYE/boardconfig.h` and `boards/ESP32_S31_MOSAICO/port/mpconfigboard.h`; `platform/ev_transport_config.h` checks backend constraints.

Distinguishing USB-Serial/JTAG from USB-OTG

P4-EYE can expose USJ and OTG CDC simultaneously. Both may appear as `ttyACM`, whose OS numbering is unstable. Identify interfaces through USB descriptors, VID/PID, serial number and board topology, then confirm device identity through `hello`/`capabilities`.

USJ and TinyUSB CDC use different peripherals/drivers. A hello response returns only to its actual ingress link, without duplication to the default interface. If one interface owns the session, the other can discover the device but acquisition returns busy.

The current P4-EYE runtime CDC PID is `0x4003`. Some S3 builds use `0x1001` to select the esptool USB reset sequence; that Serial/JTAG-like PID does not give TinyUSB OTG CDC JTAG hardware capabilities. ROM mode may enumerate a different node, invalidating the original handle.

Link = backend + slot + generation

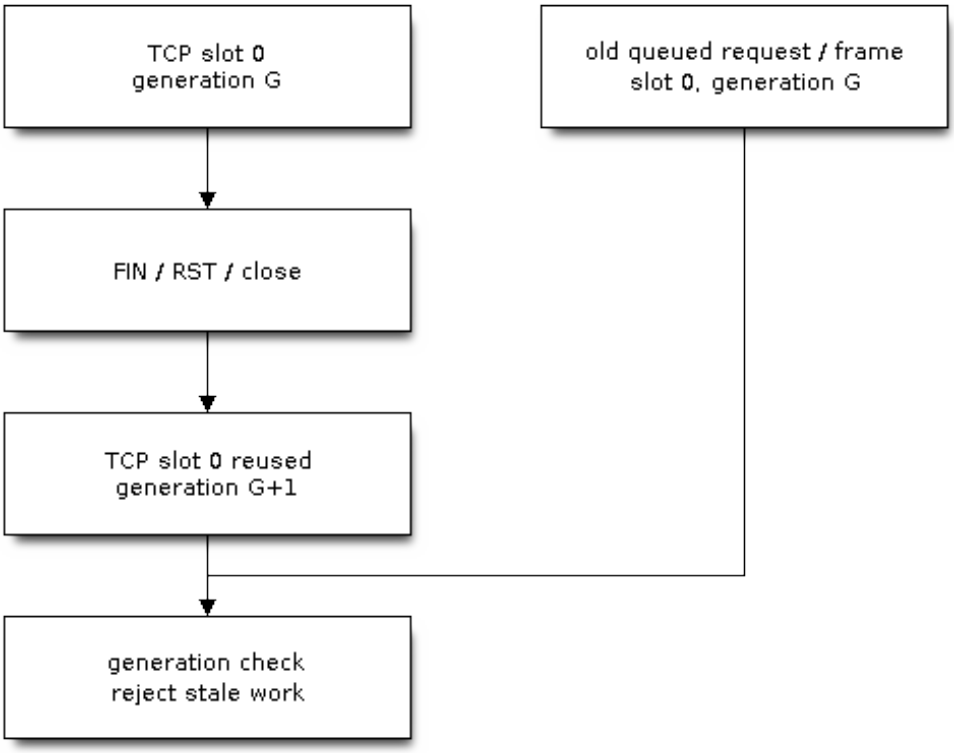


Figure 7: Reusing a TCP slot does not preserve the previous connection. Sessions bind exact link references. TX targets, queued VM requests and asynchronous callbacks must not deliver old work to a new connection. Actual USB disconnection and explicit release establish corresponding invalidation boundaries.

Board configuration differs from IDF console configuration

Table 5: Configuration layers

Configuration	Purpose
`CONFIG_ESP_CONSOLE_*`	Underlying ESP-IDF system console, startup and exception output routing; does not directly determine EV-MUX ownership.
`MI-CROPY_HW_ESP_USB_SERIAL_J`	MicroPython/ESP-VISION USJ compilation and initialization.
`MICROPY_HW_USB_CDC` / MSC port macros	Existing TinyUSB CDC/MSC composition; no second USB driver or descriptor owner is installed.
`MI-CROPY_HW_ENABLE_UART_REPI`	Whether UART REPL becomes a registered serial backend.
`ESP_VISION_TRANSPORT_DEFA` /`ESP_VISION_TRANSPORT_FAL`	Default/fallback serial output without an owner. Must be enabled `usj`, `usb` or `uart`; cannot default to unconfigured TCP.
`ESP_VISION_ENABLE_BACKGRC`	Background Wi-Fi/TCP service policy; WLAN hardware, native/Hosted driver and sdkconfig requirements are separate.
`ESP_VISION_WIFI_DEFAULT_S` /`ESP_VISION_WIFI_DEFAULT_`	Optional factory Wi-Fi credentials, empty by default; used only when no NVS record exists.
`ESP_VISION_ENABLE_TCP_TRA` / `ESP_VISION_TCP_PORT`	Board TCP backend switch and listening port, normally 3232.

The backend registry is compile-time defined, with no runtime plugin-registration API. USB connection, network service and TCP slot state remain dynamic. `supported`, `present`, `ready` and `owner` mean implementation availability, known physical presence, traffic readiness and session ownership respectively; they are not interchangeable.

`console` / `null` (alias `none`) are local output options in `ev_channel.c`, not public registered sinks in `capabilities.sinks`. Do not use them in v4 acquisition requests.

When no session owns USB CDC, default/fallback output may fill its FIFO while the host reader is closed even though USB remains mounted. A frame already partly transmitted is retained in the bounded TX queue until the host drains it or the physical link generation changes. Later frames cannot overtake it. Unstarted frames still expire, other endpoints keep running, and session-owned frames retain their transmission deadlines. This keeps a camera startup log from leaving an incomplete frame for the next connection.

Why UART cannot assume USB preview sizes

UART application communication starts at 115200 bps. With `capabilities.uartBaud` support, Device Core tests 2000000, 1500000, 921600, 460800 and 115200 in that order and keeps the first verified rate. Both the device and host port must switch; USB-OTG, USB-Serial-JTAG and TCP do not run this UART negotiation. See *Frame format, logical channels and RPC protocol* for prepare/probe/commit and rollback. ROM/stub download speed is negotiated separately during flashing.

UART estimates its transmission budget for the entire frame using the current runtime baud rate:

```
timeout_ms = max(1500, ceil(frameBytes * 10 * 1000 / baud) * 2 + 250)
timeout_ms <= 10000
```

The factor 10 represents the bits per byte for typical 8N1. Frames exceeding a 10-second budget are rejected before submission. Supporting `preview.frame` does not guarantee arbitrary JPEG delivery at low baud rates. Increase baud rate, reduce image resolution/quality or use USB/TCP; still handle TX errors and short writes.

Three distinct TCP time budgets

1. **Initial handshake**: a connection that has never acquired a session closes after approximately 3000 ms. TCP clients should not perform the serial 300 ms startup drain or lengthy UI work first.

2. **Session lease**: 15000 ms by default; absence of authorized activity releases the owner. This reclaims ownership.
3. **Idle reclamation after ownership**: a socket that lost ownership has approximately 30000 ms to reacquire before its exact-generation slot is reclaimed. This reclaims connection capacity.

Three slots support two read-only preview observers alongside one controller; see *Preview, debug capture, backpressure and statistics*. A TCP contender closes after its acquisition error is sent. The same link can reacquire within the allowed window after normal release.

DTR/RTS, RESET and ROM download

DTR changes do not acquire or release EV-MUX sessions. Closing a host serial handle while USB remains connected may be invisible to firmware. Send `session.release` before closing, or wait for lease expiry.

CDC DTR/RTS automatic download is a separate USB control path. It shares `esp_vision_restart(download_mode)` with Python `machine.bootloader()`. Python can request an immediate restart; USB control callbacks record state and a deferred timer restarts outside the callback. P4 and S31 preserve other system-register bits when setting/clearing download bits; S3 follows its ROM USB persistence/PHY handoff contract. S31 Korvo and Mosaico enable this CDC entry, while Chatbot retains its connected USB Serial/JTAG interface and does not enable its unconnected OTG port. Formal 2026.10.23 S31 firmware predates this support and requires hardware ROM entry.

The firmware change provides S31 runtime-to-ROM entry. Returning to the application after programming also requires a host that clears and verifies the S31 download latch, such as Device Core 0.7.0; older Device Core 0.6.1 does not provide that step. Publish and adopt the matching Core in IDE/website clients before releasing this firmware.

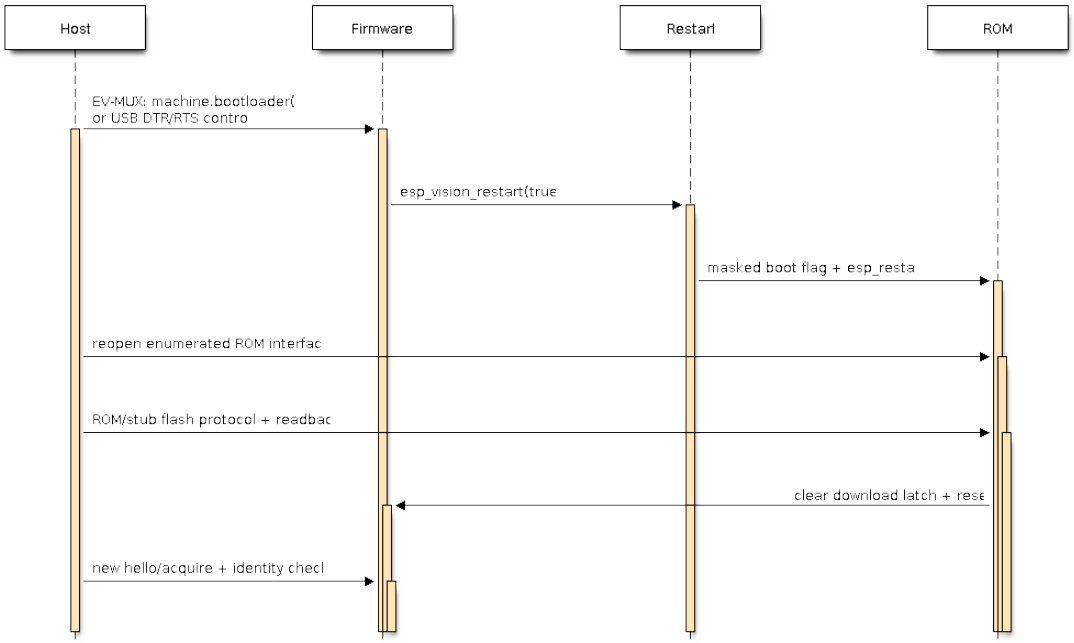


Figure 8: Entering download mode, writing images and restoring runtime require three separate confirmations. EV-MUX does not carry ROM flashing commands. Loss of its port alone proves neither ROM readiness nor successful flashing.

For P4-EYE OTG runtime-to-ROM transitions, esptool v5 uses `--before usb-reset`. Once in ROM, or retrying after re-enumeration, use `--before no-reset`. v4 spellings are `usb_reset` / `no_reset`. Verify writes, full readback, cleared download bits and runtime identity; an `EIO` alone is not success.

9.2.3 USB Interfaces, Configuration and Disk Names

This chapter describes USB configuration and the names visible on the host. See *Physical interfaces, connection generations and build configuration* for EV-MUX connections and *Storage, Filesystems and Data Protection* for recovery, backups and write ownership.

Physical interfaces and functions

Table 6: USB functions

Interface	Host function	Configuration and limitations
Serial/JTAG (`usj`)	Serial/debug and ROM download access.	Controlled by the SoC and `MICROPY_HW_ESP_USB_SERIAL_JTAG`. It provides no MSC disks. TinyUSB strings do not rename this hardware interface.
OTG CDC (`usb`)	Serial interface for EV-MUX, Python and native file transfers.	Requires the native USB device connection, `MICROPY_HW_ENABLE_USBDEV` and `MICROPY_HW_USB_CDC`. A working USJ connection does not establish an OTG connection.
OTG MSC	One mass-storage interface with one or two LUNs.	Requires `MICROPY_HW_USB_MSC` and usable storage; shares the existing TinyUSB device with CDC and enabled UVC. It does not mount a second filesystem.
OTG UVC	Camera/video function on supported builds.	Controlled by `MICROPY_HW_USB_UVC` and the existing descriptors/stream. Enumeration does not establish camera initialization or frame delivery.

Select the physical connector using the board schematic. Enumerate descriptors, select an explicit port and confirm board/firmware identity. P4 boards may expose USJ and OTG simultaneously; COM/ttyACM numbers can change after reset or ROM entry. Download-mode handoff is described in *Physical interfaces, connection generations and build configuration*.

Configuration locations

Table 7: Configuration ownership

Location	Settings
`boards/<BOARD>/port/mpconfigboard.h`	Device/CDC/MSC/UVC feature macros, USJ enablement, manufacturer/product/interface strings and board VID/PID overrides.
`boards/<BOARD>/boardconfig.h`	EV-MUX default/fallback sink and storage automatic-mount/initialization policies. Choosing a sink does not enable a USB class or connector.
Board `sdkconfig.*` and shared port defaults	IDF console, PHY/driver and FatFs settings. `CONFIG_ESP_CONSOLE_*` neither selects an EV-MUX owner nor provides MSC on USJ. `CONFIG_FATFS_USE_LABEL` enables the shared volume-label API.
`overlay/micropython/shared/tinyusb/`	Shared composite descriptors and class/endpoint configuration. Keep one TinyUSB driver/descriptor owner instead of adding another stack for disks.
`platform/usb_msc.c` and `platform/storage.h`	Fixed LUN mapping and medium names. MSC inquiry products are `ESP_FLASH` / `ESP_SDCARD`; `MICROPY_HW_USB_MSC_INTERFACE_STRING` retains each board's original name without enforcing a common prefix.

Edit board/overlay sources, not the generated MicroPython copy, and rebuild with the repository-root `idf.py --board <BOARD> ...` entry point. VID/PID changes must match the actual USB functions and host-driver contract; copying a Serial/JTAG PID does not add JTAG hardware.

Volume labels, LUNs and drive letters

Table 8: Disk identity

Medium	Python mount	Default FAT label / inquiry product	USB LUN
Internal Flash	`/` and alias `/flash`	`ESP_FLASH`	0
SD card	`/sdcard` by default	`ESP_SDCARD`	1, when SD support is compiled

The FAT label is stored on the medium and is normally displayed by the host file manager. During automatic mounting, firmware assigns a default only to an already mounted volume with an empty label. Existing nonempty labels, including custom SD labels, are preserved. Renaming never formats the disk. Writer denial or label I/O failure produces a warning without crashing or converting a successful mount into a format request. Explicit `storage.retry()` and writer-handoff refresh remain mount-only and do not change labels.

FAT labels have at most 11 OEM bytes, exclude dots and convert lowercase to uppercase; long filenames do not extend labels. Therefore `esp-vision.flash` / `esp-vision.sdcard` cannot be FAT labels; use `ESP_FLASH` / `ESP_SDCARD`. See the FatFs label API.

USB manufacturer/product/interface strings identify the device or function. A SCSI inquiry product identifies a LUN and has a 16-byte field. Neither sets the FAT label. Windows assigns `D:/E:` drive letters; other systems choose their own mount paths. Firmware cannot reserve these letters. Identify disks using the physical device, LUN and volume metadata, rather than a remembered letter or label alone.

Windows File Explorer normally names a volume using its filesystem label. The Function EV Flash volume previously had an empty label, so Explorer displayed the generic USB-drive name. `ESP32-P4X-Function-EV-Board Flash` is a USB MSC interface string describing the single interface containing both LUNs; it does not name each volume. SCSI inquiry products identify disk devices and do not set volume labels either. Assigning `ESP_FLASH` / `ESP_SDCARD` to mounted, unlabeled volumes distinguishes the media while retaining existing custom labels. An empty SD slot has no readable FAT label, so Windows may still display a generic USB-drive name; an interface string

alone cannot assign an Explorer volume name to that slot. See Microsoft's Naming a Volume and USB interface descriptor.

Unavailable media report not ready in their fixed slots. SD does not become LUN 0 if Flash fails, and `/flash` adds no disk. Boards without SD support expose only Flash. Empty-slot visibility and drive-letter allocation are host behavior.

Mounting, ownership and validation

Flash and SD automatic mounting are independent and enabled by default. Formatting an unrecognized filesystem is separate: the default board policies are `ESP_VISION_STORAGE_FLASH_FORMAT_IF_NO_FILESYSTEM=1` and `ESP_VISION_STORAGE_SD_FORMAT_IF_NO_FILESYSTEM=0`. Disk naming never overrides these policies or authorizes formatting; see *Storage, Filesystems and Data Protection*.

An EV-MUX owner does not automatically own MSC writes. Safely eject the selected OS volume and complete the per-volume handoff in *Interactive Python, scripts and file operations* before Python/IDE writes or backup. Two disks do not permit simultaneous host/VM writes to either shared FAT filesystem. Keep verified backups outside the device.

1. Inspect the selected device and `capabilities.sinks`. Connect the native USB device port for MSC acceptance; formatting a card cannot fix an absent OTG connection.
2. Check `storage.status("flash")` / `storage.status("sd")`. Mount and host writer admission must both succeed before the corresponding LUN is ready.
3. Inspect USB functions and logical disks separately. Record physical serial, firmware, LUN, actual volume label and OS drive letter. A COM port or product string alone is not evidence of a mounted disk.
4. Verify isolated test files/hashes and persistence, preserving existing data and custom labels. Unavailable media should return diagnostics while the other volume/control link remains usable.
5. Report CDC, actual dual-LUN enumeration and healthy SD read/write separately. USJ-only acceptance does not cover native USB MSC or both disks.

9.2.4 Session ownership, acquisition and recovery

What takeover means in v4

All supported physical interfaces can compete for acquisition, but only the first valid request accepted by the device becomes owner. Later requests cannot force out a valid owner. Default ports, DTR, USB insertion order and debug streams do not establish priority.

`route.bind` and `route.auto` are no longer public operations and return `UNSUPPORTED_OPERATION`. Both user/debug streams transfer with the sole owner. For host B to take over, A must release, or its link/lease must expire, before B acquires.

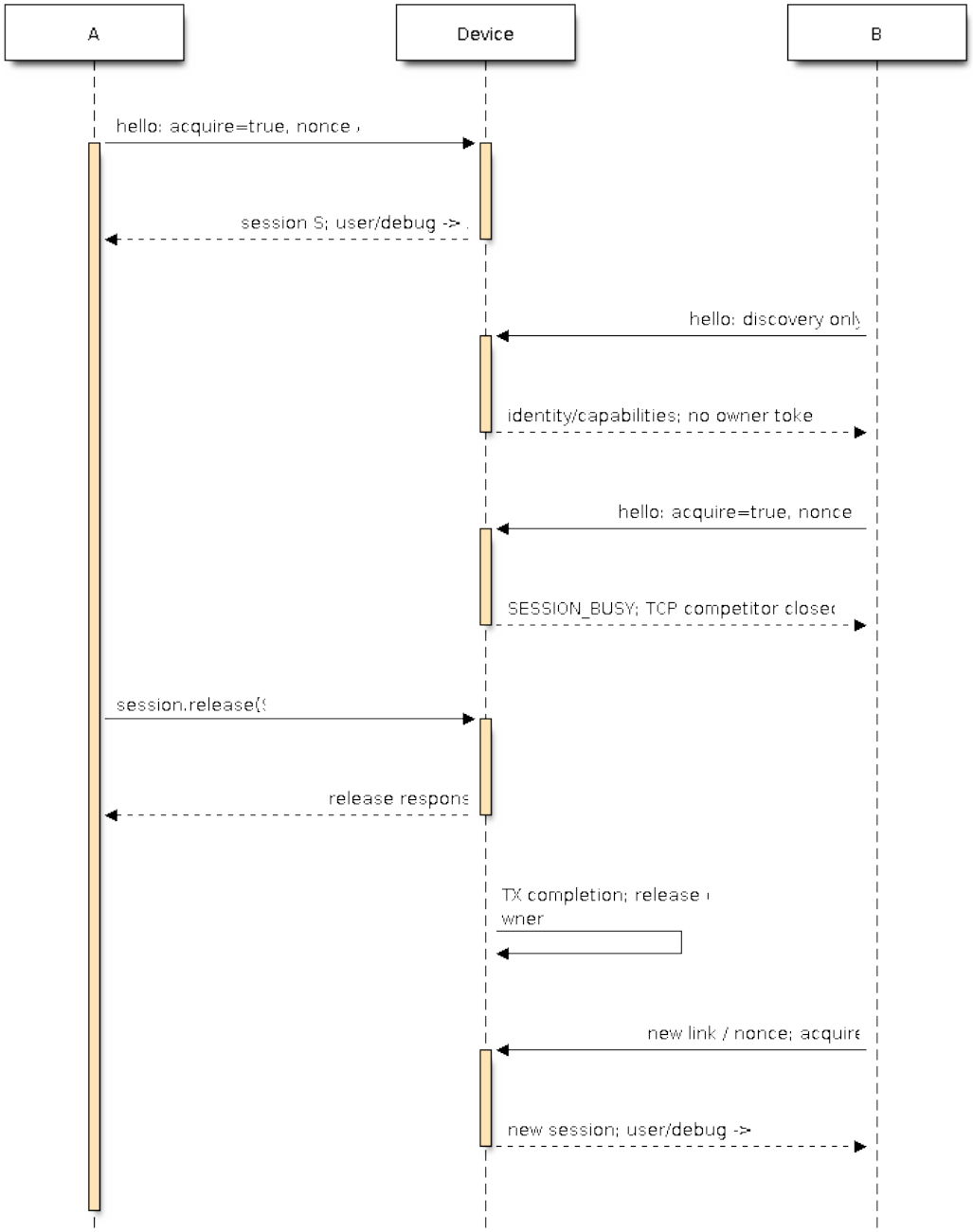


Figure 9: Discovery, failed competition and explicit handoff. Discovery does not authorize B to send Python/file commands, and A cannot transfer its token to B.

Session state machine

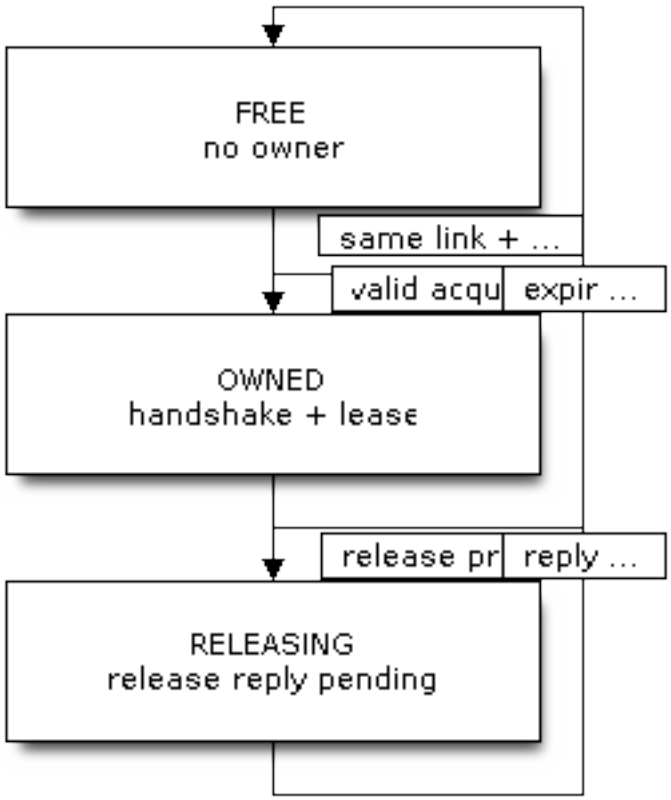


Figure 10: Release preparation and completion preserve the link until its reply is sent. Immediate reacquisition after receiving release may briefly return `SESSION_RELEASING`. On the same link, retry the same hello nonce within a bounded total deadline; never retry indefinitely.

Initial discovery and acquisition

Drain startup residue after opening serial; TCP acquires directly. Metadata and payload in the following block are two parts of one frame, not separate JSON frames:

```
{"seq":1,"channel":"user.rpc","type":"req","method":"hello","encoding":"json"}

{"evMuxVersion":4,"acquire":true,"clientNonce":"0123456789abcdef0123456789abcdef"}
```

Successful payload example with additional capability fields omitted:

```
{
  "ok": true,
  "evMuxVersion": 4,
  "session": {
    "id": "89abcdef0123456789abcdef01234567",
    "owner": "tcp",
    "leaseMs": 15000,
    "keepaliveMs": 5000,
    "vmEpoch": 1
  },
  "vmReady": true,
  "routes": {"user": "tcp", "debug": "tcp"}
}
```

Match `rsp/hello` by seq, channel and method before accepting the new token. Empty hello/capabilities only discover the device. Discovery replies return to the exact ingress link and do not disclose the valid owner ID to contenders.

`clientNonce` is a 32-character lowercase hexadecimal identifier for one logical acquisition. Reuse it only when retrying that acquisition on the same link; generate a new value for a new socket, reopened serial port or new acquisition. Same-link, same-nonce idempotent hello can return the same valid session, not transparently replay arbitrary operations.

Handshake completion is gated on acquisition-reply TX completion. Generating a sessionId internally does not authorize the host to submit dependent operations before receiving a valid reply.

Keepalive is not application completion

```
{"seq":2,"channel":"user.rpc","type":"req","method":"session.keepalive","encoding":
↪"json","sessionId":"89abcdef0123456789abcdef01234567"}

{}
```

Default keepalive is 5000 ms and lease 15000 ms; authorized application activity also renews the lease. Schedule according to reported values and keep reading/renewing throughout long scripts, file transfers and preview. Do not wait for an operation to finish before renewing.

Successful keepalive confirms current link/session validity and can report VM generation/readiness. It does not prove script completion, fully published files or applied network configuration.

Four identity/correlation dimensions

Table 9: Non-interchangeable fields

Dimension	Purpose	What it cannot imply
link generation	Reject work left over from an old physical connection.	The same port name or TCP slot need not represent the same connection.
`sessionId`	Authorize the exact link owning the current session.	Cannot transfer from discovery replies, another host or an old socket.
`vmEpoch`	Identify generations of VM operations and output.	Advancing generation does not imply an idle/executable VM or REPL.
`seq` + channel + method	Correlate a response within one request context.	An event, preview or old response with the same seq cannot complete this RPC.

Except for read-only TCP preview subscriptions, all non-discovery operations carry sessionId. VM-scoped `file.write`/`script.write`/`script.run`, camera/object/VFS debugging and `repl.stdin`/`repl.signal` also carry current vmEpoch. Transport statistics, keepalive, release and background network controls do not require an idle VM, but still require appropriate owner authorization.

Startup VM readiness and soft reset

Current initial `vmEpoch` is a random nonzero uint32; older v4 builds may report 0 during startup. Transport controls retain the returned value; VM operations wait for a positive epoch and `vmReady:true`. Nonzero sequence numbers wrap from 4294967295 to 1. Compare using wrap-aware ordering rather than simple numerical greater-than.

Soft reset may preserve sessionId and networking; epoch advances at reset start. `event/hello` is supplemental, not sufficient proof. Confirm the new epoch/readiness through a correlated keepalive, then perform a current-generation REPL barrier requiring an actual new marker.

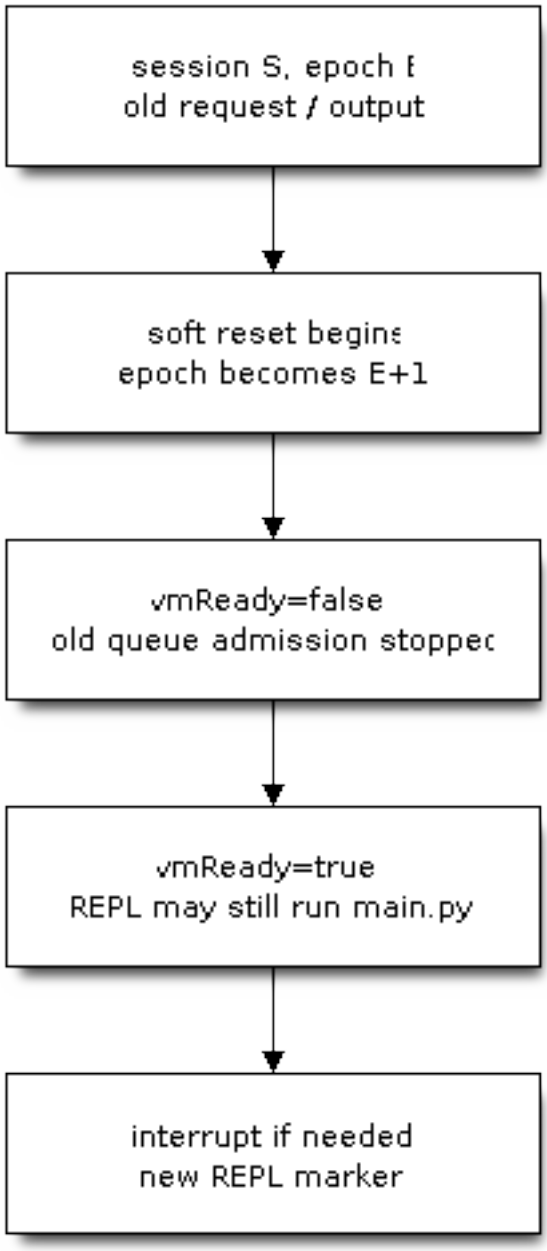


Figure 11: Transport liveness, VM readiness and executable REPL are separate conditions. Restarted `boot.py`/`main.py` may still occupy REPL; readiness alone does not permit immediate VFS/camera VM RPCs.

Hard reset/ROM entry invalidates old tokens, epochs, pending requests and decoder state. Rediscover/acquire after runtime returns. Soft reset clears old VM output parsing state; it must not arbitrarily discard an in-progress physical EV-MUX frame buffer.

Disconnects, FIN/RST and silent disappearance

- TCP FIN/RST or confirmed invalid link: reclaim its owner and invalidate old-generation output; create a new connection and nonce before acquisition.

- Actual USB disconnection invalidates the link. Closing a host serial handle or changing DTR alone may not notify firmware; release explicitly.
- Silent USJ/UART disappearance: wait for lease reclamation. Port type and default/fallback status do not grant early acquisition rights.
- A formerly owned TCP link may retain a 30-second idle window before slot reclamation; this differs from the 15-second lease.

Lease release neither reverses completed external side effects nor guarantees the previous program stopped. After acquisition, B should confirm VM state, interrupt explicitly when necessary, and verify executable REPL with a new marker.

Errors and recovery decisions

Table 10: Common errors

Error	Host response
`SESSION_BUSY`	Another valid owner exists; request release/wait rather than forcing takeover or reusing its token.
`SESSION_RELEASING`	Same-link release completion is pending; retry the same nonce only within bounded acquisition.
`SESSION_REQUIRED` / `NOT_OWNER`	Stop old-session operations, discard local session state and establish a new session.
`VM_EPOCH_MISMATCH`	The original VM request is invalid; reconfirm generation and do not automatically replay scripts/file writes.
`VM_BUSY` / `VM_TIMEOUT`	VM work cannot queue or was not serviced in time; resolve running scripts, cancellation/reset and actual REPL liveness.
Link TX failure / host timeout	Side effects may already exist. Mark completion unknown and query/verify file or transaction state first.

Network transactions with the same `operationId` have dedicated correlation/recovery semantics. `seq` is not a universal idempotency key. Scripts, REPL input and file append have no general transparent-replay contract.

9.2.5 Frame format, logical channels and RPC protocol

Three version identifiers

`firmware.version` identifies the Git/release build; `evMuxVersion` defines the required session protocol; `EVMUX/1` identifies wire framing. Determine compatibility from evMuxVersion/capabilities, then display, record or verify firmware.version. Do not infer protocol from date strings.

Read-only firmware installation evidence

`firmware.status` is a `user.rpc` method advertised as `firmware.status.v1` only by the opt-in `ESP32_S31_MOSAICO` Iris-package build. Acquire an EV-MUX v4 owner session first. Send `{"imageSize":N}`, where N is the exact application BIN length from the validated package (1–7340032 bytes). The request runs on the transport task, without REPL entry, Python execution, writes or reset. Missing/malformed sizes return `INVALID_ARGUMENT`; incomplete VM startup returns `NOT_READY`; unconfirmed OTA or unreadable installation evidence returns `EVIDENCE_UNAVAILABLE`.

A successful response has `ok=true`, `schemaVersion=1`, `deviceId` (factory MAC as 12 lowercase hex digits), `board`, `firmware.id/version`, a 32-character hexadecimal `bootId`, `vmReady=true`, `bootConfirmed=true`, numeric IDF `resetReason` and `crashReset`. The boot token is stable across VM soft resets

and changes on hardware reboot; the installer compatibility layer uses its first eight bytes as the little-endian Iris Boot ID.

`application` contains `address`, partition `capacity`, requested `imageSize`, full BIN `sha256`, `elfSha256` and `otaState="valid"`. The full BIN hash includes all requested bytes, including the appended image digest; it is not ESP-IDF's embedded image hash. `partitionTableSha256` covers exactly 4096 Flash bytes starting at the configured partition-table offset. `installation.operationId` and signed `installation.result` expose the retained installer's persisted commit receipt without sending or implementing its protocol. Missing or malformed receipts fail the query instead of inventing a successful receipt. A recorded nonzero result is returned unchanged and must fail host acceptance.

An installation verifier must compare the live device identity, board, exact package length/hashes/layout and operation ID against its intended installation, require zero commit result and `crashReset=false`, and observe a stable boot token across acceptance. A healthy query for another image or an earlier operation is not installation success. Startup confirmation does not validate user Python code or peripherals. This interface does not enter Vibe Mode; the operator uses AI + Reset for the next installation.

Byte layout of a frame

```
0x1e + ASCII "EVMUX/1 h=<H> p=<P> c=<CRC32>\r\n"
       + H bytes of UTF-8 metadata JSON
       + P bytes of payload
       + 0x1f
```

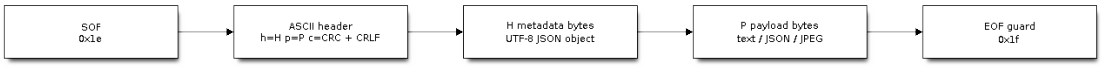


Figure 12: H/P define body boundaries; EOF is not a payload delimiter. JPEG, binary file data and text can contain `0x1e`, `0x1f` and CRLF. Never truncate the body on those bytes.

H/P count UTF-8 or binary **bytes**. A character such as U+4F60 typically occupies three UTF-8 bytes; JavaScript string length is not payload_bytes.

CRC32 covers only payload, not metadata/header. Use standard CRC32, e.g. host Python `zlib.crc32(payload) & 0xffffffff`, encoded as eight hexadecimal digits. Device frames include payload CRC; incoming `c=00000000` skips verification. CRC detects corruption, not identity or final operation completion.

For JPEG responses and previews, a valid EV-MUX frame alone does not establish a valid image. Firmware validates encoder output before queueing it; Core additionally checks the JPEG structure and dimensions before delivery. Invalid capture responses return `INVALID_IMAGE` locally and invalid previews are dropped; see *Preview, debug capture, backpressure and statistics* for the supported image checks and their limits.

Hosts must handle arbitrary fragmentation and coalescing

```
read #1: [SOF][header first half]
read #2: [header tail][metadata first half]
read #3: [metadata tail][payload first half]
read #4: [payload tail][EOF][whole next frame][next SOF]
```

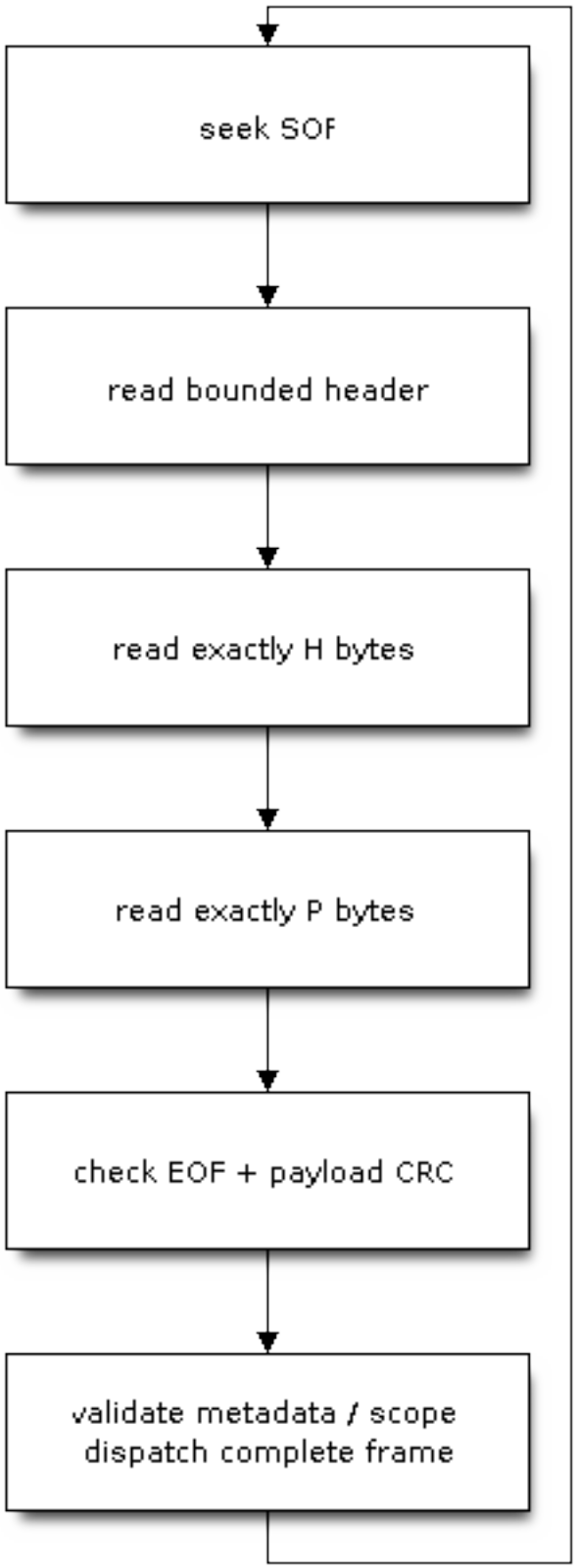


Figure 13: A streaming parser retains state across reads and delivers only fully validated frames. Record format/length/CRC/deadline errors and resynchronize in a controlled manner; never pass partial frames into REPL/file writes.

Current device RX limits are 384 metadata bytes and 2048 payload bytes. RX parser deadlines are approximately 3000 ms idle and 10000 ms per complete frame. These limit **host commands received by the device**, not preview image size or host receive timeouts for large images.

Metadata fields

Table 11: Common fields

Field	Type	Purpose
`channel`	String	Select channel, e.g. `user.rpc` or `repl.stdout`.
`type`	String	RPC request `req`, response `rsp`, notification `event` or data `data`.
`seq`	uint32	Request/response correlation number; identical numbers across channels/types do not complete each other.
`method`	String	RPC method or stdio source descriptor; `c-stdio` is not an RPC method.
`encoding`	Optional string	Semantic hint such as `json`, `text` or `binary`.
`sessionId`	32-character lowercase hexadecimal string	Current owner authorization/output scope; discovery requests omit tokens.
`vmEpoch`	uint32	VM operation/output generation scope.
`sid`	Optional string	Logical source/domain such as `stdio`; not sessionId.
`ts_ms`	Optional integer	Device timestamp, not automatically synchronized with host time.
`contentType`、`width`、`height`	Channel-defined	Describe image payload; JPEG bytes remain in payload, not JSON.

Metadata and RPC request payload must be complete top-level JSON objects with correctly typed known fields and no duplicates. Unknown extensions may be ignored. Escaped strings require a real JSON parser; trailing garbage is not a valid complete object.

Channels and streams

Table 12: Wire channels

channel	Direction	stream	Payload / purpose
`user.rpc`	Bidirectional	user	JSON discovery/session/script/device/network control.
`debug.rpc`	Bidirectional	debug	Debug RPC; capture success responds directly with binary JPEG.
`repl.stdin`	Host to device	user	Raw REPL input bytes, authorized by session/epoch with a separate size limit.
`repl.signal`	Host to device	user	Currently accepts only one byte `0x03` for Ctrl-C.
`repl.stdout`	Device to host	user	Text blocks from Python/REPL/HAL and C stdout.
`repl.stderr`	Device to host	user	Explicit stderr paths such as C stderr; Python exceptions may use stdout.
`log.idf`	Device to host	debug	IDF/platform log text.
`preview.frame`	Device to host	user	JPEG data frames produced by `img.flush()`.

Channel-to-stream assignment is fixed. Hosts do not bind separate serial ports for preview/stdout/RPC. The owner holds both streams; RPC replies return to the exact request link.

RPC methods and execution layers

Table 13: EV-ATP control plane

channel	method	Execution layer	Description
user.rpc	`hello` / `capabilities`	transport	Discovery; hello can explicitly acquire ownership.
user.rpc	`session.keepalive` / `session.release`	transport	Renew lease, query ready/epoch, release owner.
user.rpc	`transport.uart.prepare` / `transport.uart.probe` / `transport.uart.commit`	transport	UART owner baud negotiation; ordinary UART output pauses until commit or rollback.
user.rpc	`file.write`	transport / native VFS	Verified binary upload; session/VM-scoped without interpreter execution.
user.rpc	`script.write` / `script.run`	VM	Legacy Base64 writer / queue script execution.
user.rpc	`device.control`	Primarily transport	Dispatch stop/reset/status, network.*, storage.writer.*, launcher status and other actions.
debug.rpc	`transport.state` / `route.get`	transport	Observe state/routes without changing public stream bindings.
debug.rpc	`debug.info`	VM, except stats	device/memory/sensor/fs.list/fs.read; transport.stats does not require VM.
debug.rpc	`debug.capture_frame`	VM	Encode the last independently published application acquisition in this RPC rsp; no sensor initialization or physical acquisition. Requires `debug.capture_frame.observed.v1`.
debug.rpc	`debug.capture_screen`	VM	Encode the last LCD RGB565 frame as JPEG in this RPC rsp, without camera calls.
debug.rpc	`route.bind` / `route.auto`	Rejected	v4 returns `UNSUPPORTED_OPERATION`.

Device Core APIs `files.read/remove` and `python.execute` do not imply identically named wire RPCs. Their file/Python composition is described in *Interactive Python, scripts and file operations*.

`debug.capture_screen` requests use `debug.rpc` with the current session and VM epoch; optional `quality` is 1–100. Successful responses retain request seq/method, set `encoding:binary` and `contentType:image/jpeg`, include `width` / `height` and carry JPEG payload. An absent LCD returns JSON error `NOT_SUPPORTED`; no valid displayed frame returns `DISPLAY_NOT_READY`. Never substitute camera output. Core's `capture({ source: 'screen', format: 'jpeg' })` uses this RPC; see *Preview, debug capture, backpressure and statistics*.

Requests, responses and asynchronous events can interleave

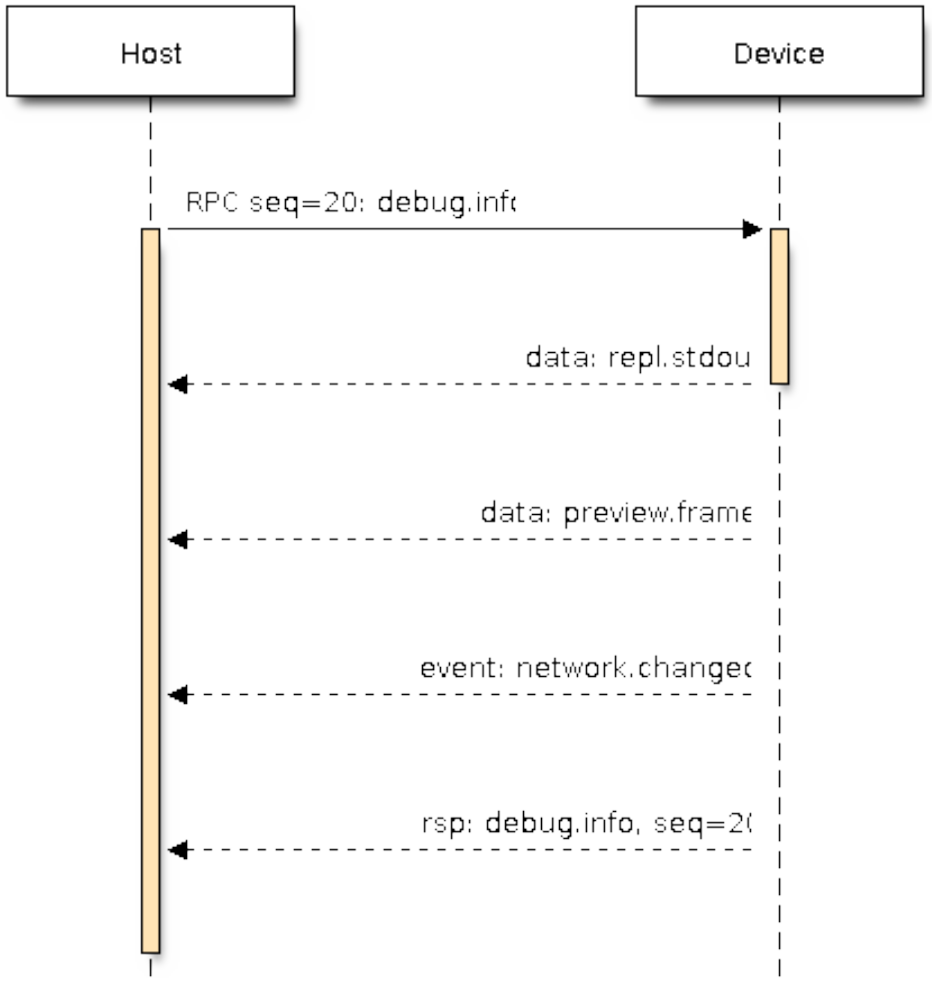


Figure 14: While waiting for seq=20, the reader must continue handling other complete frames. Match at least type, seq, channel and method, then session/epoch as appropriate. Even an identically numbered `event/hello` cannot serve as `rsp/hello`.

Acquisition replies may allocate a new session; discovery replies grant no owner authorization. Successful VM replies must match the requested epoch; control replies may report a later current epoch. Some authorization errors omit session fields: handle their codes without accepting returned values as new authorization.

Reject owner `repl.stdout`, `repl.stderr` and `preview.frame` output from other sessions or old VMs. Never attribute images/control replies to whichever request was most recent.

A verifiable host encoder

This **host CPython** example explains framing. Use the shared client in *Host integration, Wi-Fi configuration and recovery* for actual communication.

```
import json
import zlib

def frame(metadata, payload):
    meta = json.dumps(metadata, ensure_ascii=False,
```

(continues on next page)

(continued from previous page)

```
                      separators=(",", ":")).encode("utf-8")
    data = payload if isinstance(payload, bytes) else payload.encode("utf-8")
    crc = zlib.crc32(data) & 0xffffffff
    header = ("\x1eEVMUX/1 h=%d p=%d c=%08x\r\n" %
              (len(meta), len(data), crc)).encode("ascii")
    return header + meta + data + b"\x1f"

request = frame(
    {"channel": "user.rpc", "type": "req", "method": "hello",
     "seq": 1, "encoding": "json"},
    json.dumps({"evMuxVersion": 4, "acquire": True,
                "clientNonce": "0123456789abcdef0123456789abcdef"}),
)
```

This function only encodes frames. It implements no session, concurrent reader, deadlines, RPC correlation or recovery, and cannot replace a complete client. Do not assume host `zlib`/thread/socket combinations are available unchanged in device MicroPython.

Authorized file.write requests are the bounded exception: binary payloads can carry 16384 file bytes plus a 20-byte ID/offset prefix, with CRC32. All other control payloads retain the 2048-byte limit. See *Interactive Python, scripts and file operations* for begin/data/commit/abort.

Runtime UART baud negotiation

A UART-enabled firmware advertises `capabilities.uartBaud` with `baseline:115200`, `current`, `candidates:[2000000,1500000,921600,460800,115200]`, `probeBytes:512`, `timeoutMs:5000` and `switchDelayMs:100`. The field is `null` without a UART backend. These values describe application communication; ROM/stub flashing has its own protocol and lifecycle.

The authenticated UART owner sends the following transport-scoped methods on `user.rpc` with its current session ID; they remain available during a VM soft reset. USB-OTG, USB-Serial-JTAG and TCP owners cannot change the UART baud rate.

1. `transport.uart.prepare` takes `{baudRate,token}`, where the rate is a listed candidate and token is 1–32 ASCII letters, digits, underscores or hyphens. Its response includes `{ok:true,baudRate,token,timeoutMs,switchDelayMs}` at the **old** rate. Firmware pauses ordinary UART output, finishes any already-started frame, drains the hardware FIFO **and shift register**, then switches after the advertised delay. Host transports reopen/update their serial port and wait at least the advertised delay plus a guard before probing.
2. `transport.uart.probe` takes `{token,data}`, where data is 1–512 printable ASCII bytes. The response returns identical `token` and `data` plus `baudRate`. Hosts compare several different lengths and patterns in both directions; the first usable rate wins in the advertised candidate order. Probes use normal EV-MUX CRC32.
3. `transport.uart.commit` takes `{token}` after at least one probe and returns `{ok:true,token,baudRate,...}`. Ordinary output resumes after the commit response is queued to hardware. Probe/commit with the same committed token are idempotent, allowing a host to recover a lost commit acknowledgement.

An unfinished negotiation expires after 5000 ms, releases its owner, restores 115200 and clears UART RX/parser state. After a successful negotiation, session release or lease expiry also restores 115200; a lost final commit response may therefore require recovery through the 15000 ms session lease. Hard reset starts at 115200. VM soft reset preserves the physical rate and session lifecycle. Firmware without this capability stays at the initial rate; no host should send speculative baud commands to it.

9.2.6 Interactive Python, scripts and file operations

One interpreter, multiple host operations

Friendly REPL, script execution and file queries share one MicroPython VM. Binary upload writes native VFS directly under the same session, VM generation and storage drain gate; it does not execute Python. Core still serializes file and interpreter operations. A long-running program affects REPL and VM RPCs, but does not automatically stop transport keepalive/network controls.

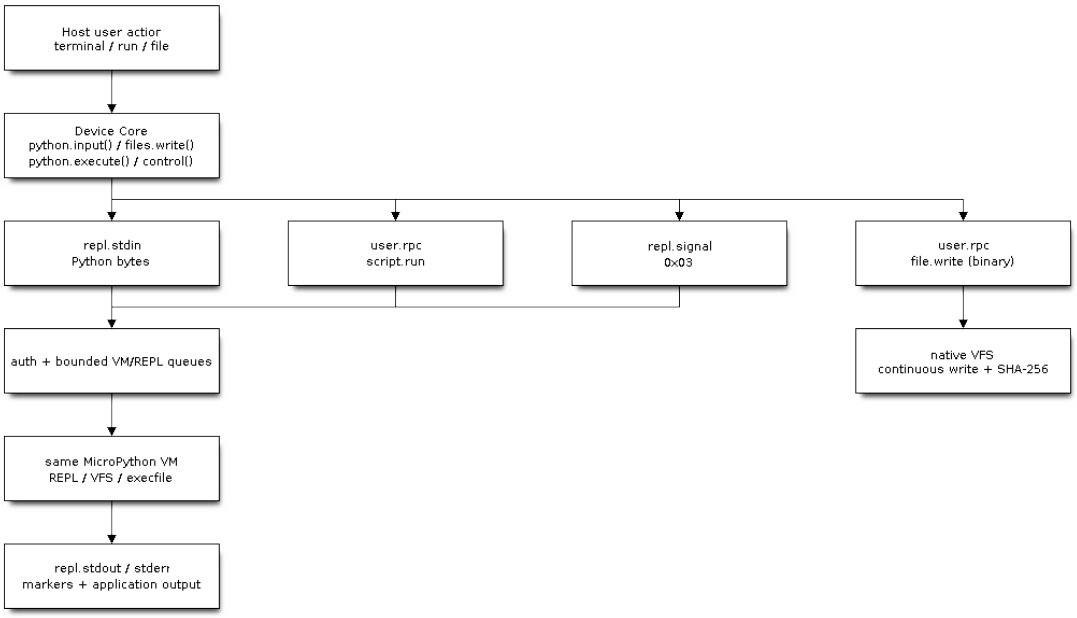


Figure 15: Interactive input, file commands and run commands are separate entry points, without separate interpreter isolation. The session owner must coordinate them; two callers cannot safely concatenate simultaneous REPL commands.

Interactive Python: framed stdin

The following metadata/payload form one frame. Token/epoch are placeholders and must come from this acquisition:

```
{"seq":10,"channel":"repl.stdin","type":"data","encoding":"text","sessionId":
↪"89abcdef0123456789abcdef01234567","vmEpoch":1}
```

```
print("hello from current VM")
```

Follow Python text with appropriate REPL line endings, e.g. `\r\n`. `repl.stdin` carries Python bytes, not JSON `{"code":"..."}`. This channel is not RPC and does not return a matching `rsp` for every input; hosts correlate execution markers in REPL output.

Capabilities normally report `replInputMax` as 511 bytes, the usable capacity of a 512-byte ring. Limit each input frame by actual encoded length: multiline input, Unicode, emoji, escapes and CRLF all consume bytes. The 2048-byte control-payload limit does not apply to stdin.

Device Core constructs Python literals using ASCII escapes and places multiline source in `exec(...)`, avoiding mismatched UTF-8/source input budgets. **This is client encoding policy, not firmware automatically escaping arbitrary oversized source.**

Ctrl-C, Ctrl-D and stopping programs

Ctrl-C must use `repl.signal` with exactly one payload byte `0x03`:

```
{"seq":11,"channel":"repl.signal","type":"signal","sessionId":
↪"89abcdef0123456789abcdef01234567","vmEpoch":1}
```

```
payload hex: 03
```

With EV-MUX enabled, raw `0x03` on physical serial is invalid/incomplete framing, not Ctrl-C. Text "03" and JSON `3` are not that byte.

Friendly REPL Ctrl-D (`0x04`) can arrive through stdin to trigger soft reset; Device Core can also request it through framed Python. `device.control` action `stop` provides a control-plane stop entry, but acceptance does not prove interrupted code has exited.

After stopping, require a new complete marker in the current session/epoch to prove executable REPL. Ctrl-C can be delayed by native calls that do not return promptly. A single RPC ack does not prove camera/file resources were released.

Completion markers and command echo

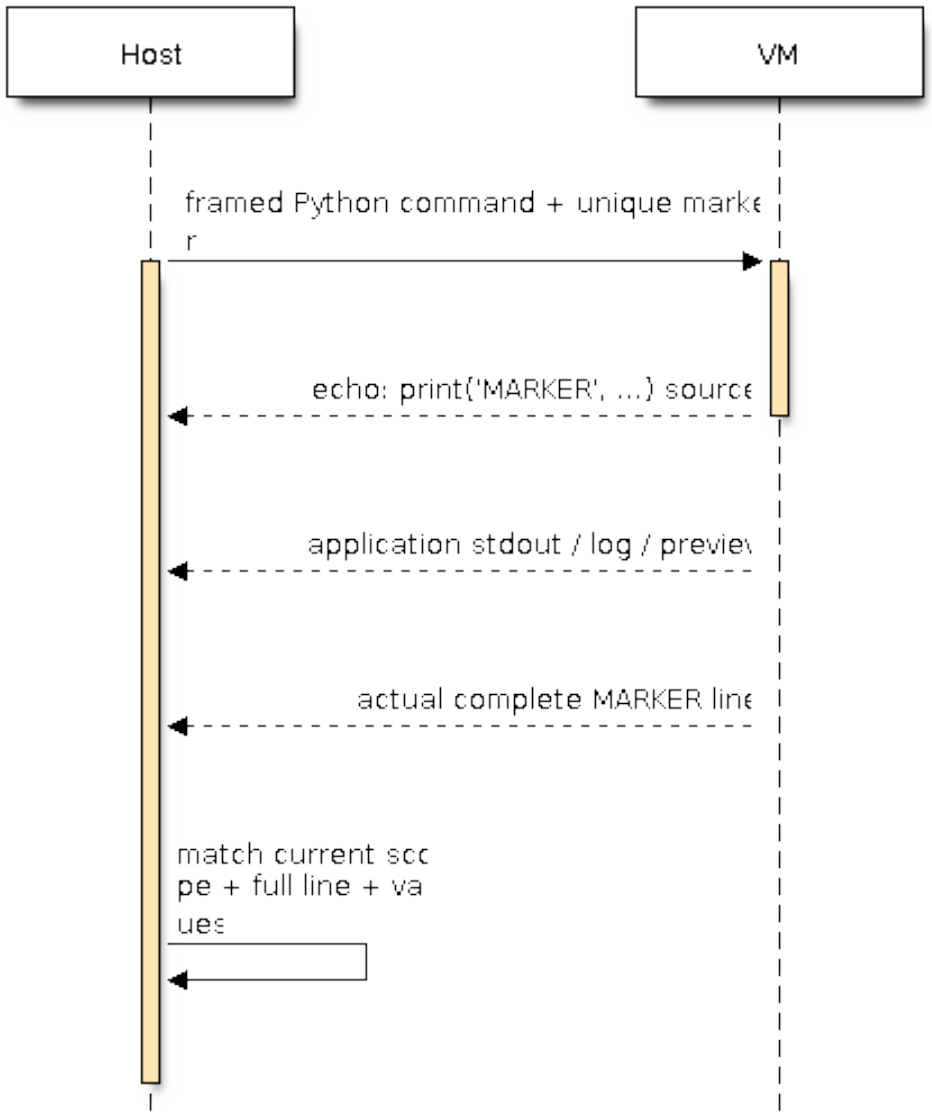


Figure 16: Echo includes the marker string; only the later executed print line proves completion. Reassemble complete lines across frames. String contains matching alone confuses echo with success.

Use a unique marker per run, current sessionId/vmEpoch, complete-line structure and expected result. Camera tasks also need actual image dimensions or other application evidence. Exceptions can appear on stdout; empty stderr is not proof of success.

Reliable binary file upload

SDK 0.2.0 uses one upload engine for Python, model, image and arbitrary binary files. Large Python execution stages its temporary source through this engine before `script.run`. Public `files.write` keeps its signature. Older v4 firmware without `file.write.binary` can still be inspected, but upload requires updating firmware; Core never silently switches back to the slow legacy writer. The firmware retains script.write only for previously released clients.

All operations use method `file.write` on `user.rpc`, current sessionId/vmEpoch and request/response seq correlation. Small JSON controls remain capped at 2048 bytes. The host creates a fresh 16-byte random upload ID before begin, so cancellation or a lost begin response can still issue abort for that ID.

```
{"action":"begin","path":"/models/example.tflite","mode":"overwrite","totalBytes
↪":2097152,"sha256":"<64 lowercase hex>","uploadId":"<32 lowercase hex>"}
{"ok":true,"uploadId":"<same ID>","chunkBytes":16384}
```

Data requests set metadata `encoding:binary`. Their payload is 16 raw upload-ID bytes, a 4-byte little-endian offset, then 1..16384 raw file bytes. CRC32 covers the complete payload, including the prefix. Non-final blocks are multiples of 4096 bytes. The final block can be shorter. NUL, CRLF, SOF and EOF bytes are ordinary payload; there is no Base64 or Python input interpretation. The response is JSON `{ok:true,writtenBytes:N}`, checked before sending the next block. Duplicate, skipped or out-of-order offsets reject; this is sequential upload, not random access or automatic replay/resume.

```
{"action":"commit","uploadId":"<same ID>"}
{"ok":true,"bytes":2097152,"complete":true}
{"action":"abort","uploadId":"<same ID>"}
```

The device keeps one temporary native file open for the entire transfer. An admitted VFS operation remains held until close, preventing MSC handoff across a live descriptor. It checks length and a streaming SHA-256 against begin metadata, fsyncs, closes and publishes only after verification. Empty files use begin/commit without data frames. The receiver allocates larger frame storage only after verifying a file.write binary request from the active owner; limits for unrelated control RPCs remain unchanged. Raw payload is limited to 16384+20 bytes, independent of the 2048-byte JSON limit.

The upload is tied to the physical link generation, session identity, VM epoch and transition generation. Old owners cannot append to or abort a newer upload. Abort, stale generation, session loss or 30 seconds without file progress close/remove the temporary file; failed close retains the storage drain barrier rather than claiming a clean handoff. Uploads are invalidated on VM reset even though native I/O needs no interpreter execution. No file buffer is permanently reserved at boot.

Existing target data is preserved during transfer and integrity checks. FatFS cannot replace a destination in one rename: replacement retains the old regular file as a backup until the verified temporary file is renamed, and attempts rollback on ordinary publication failure. Power loss between FAT directory updates can leave staging/backup files and is not a journaled atomic replace. A successful commit followed by lost response has unknown completion from the host's perspective; inspect the destination before retrying. SHA-256 detects transport/content mismatch against host input, not malicious-host authenticity.

Upload design and measured performance

A file is a byte sequence; its extension does not change transfer semantics. Upload and execution are separate operations, so Python, model and image data share one binary engine. Per-transfer open/hash/close replaces per-chunk open/append/close. Raw 16 KiB blocks reduce framing and round trips, and non-final blocks align to Flash's 4 KiB

erase size. This reuses existing EV-MUX framing and ownership without another physical connection, public upload API or transfer window.

On 2026-10-16, P4X-EYE revision 3.2 with ESP-IDF 6.1 firmware `af41759` and SDK 0.2.0 development sources produced the following application upload times on the existing Flash filesystem. Times include begin/data/verified commit; separate on-device full-file SHA-256 readback passed after every upload. These are observed samples, not board-independent throughput guarantees.

Table 14: P4X-EYE Flash upload samples

Interface	256 KiB legacy writer	256 KiB binary upload	2 MiB binary upload
USB-Serial-JTAG	40.94 s	7.24 s	55.75 s
USB-OTG	40.75 s	7.01 s	55.55 s
TCP / Wi-Fi	92.33 s	9.26 s	62.17 s

The 256 KiB transfer dropped from 292 data RPCs to 16 data RPCs plus begin/commit. TX+RX application bytes fell from roughly 2.14 to 1.04 times file size. The local Flash/FAT sequential-write baseline was about 55 KiB/s; it is a baseline for the current storage implementation, not a physical USB bandwidth limit. Small files still pay fixed begin/commit and FAT metadata costs. Improving further requires measuring storage and receive scheduling rather than increasing JSON chunk sizes.

Public file API and wire implementation

Table 15: File interface layers

Host operation	Wire implementation	Use case
`device.files.write({ path, data, overwrite, ... })`	Core uses file.write control requests and raw binary blocks on user.rpc, with continuous native VFS writes and verified commit. See the binary contract above.	The application upload entry for strings and binary bytes. It does not automatically run Python.
Device Core `EvRepl.writeFile()`	Framed REPL Python `open` / `binascii.unhexlify` / `write` / `rename`.	Advanced low-level helper, not the current application `files.write()` path or a firmware RPC named writeFile.
`device.files.read()` / `list()` / `remove()` / `mkdir()`	Framed REPL `seek/read` / `os` queries/deletion with controlled output parsing.	Larger files, result validation and cancellation; still requires executable VM/REPL.
`debug.info` scopes `fs.list` / `fs.read`	`debug.rpc` with bounded JSON/base64 debug replies.	Quick diagnostics; current fs.read reads at most the first 384 bytes, not a complete paginated offset download API.

Legacy script.write requests and chunks

The following contract is retained for SDK 0.1.x clients only; SDK 0.2.0 never uses it.

```
{"seq":20,"channel":"user.rpc","type":"req","method":"script.write","encoding":
↪"json","sessionId":"89abcdef0123456789abcdef01234567","vmEpoch":1}

{"path":"/hello.py","encoding":"utf-8","mode":"overwrite","offset":0,"totalBytes
↪":11,"contentBase64":"cHJpbnQoMTIzKQo="}
```

Example content is 11 bytes: `print(123)\n`. Generate source bytes first and use `len(source_bytes)` for totalBytes; the example below calculates rather than guesses the length.

```
import base64
source = b"print(123)\n"
payload = {
    "path": "/hello.py", "encoding": "utf-8", "mode": "overwrite",
    "offset": 0, "totalBytes": len(source),
    "contentBase64": base64.b64encode(source).decode("ascii"),
}
```

`mode` accepts `overwrite` / `create` and `encoding` accepts `utf-8` / `binary`. Provide offset/totalBytes together or omit both. Base64 expands data by about 4/3, plus path/JSON overhead: a 2048-byte control payload does not fit 2048 raw file bytes.

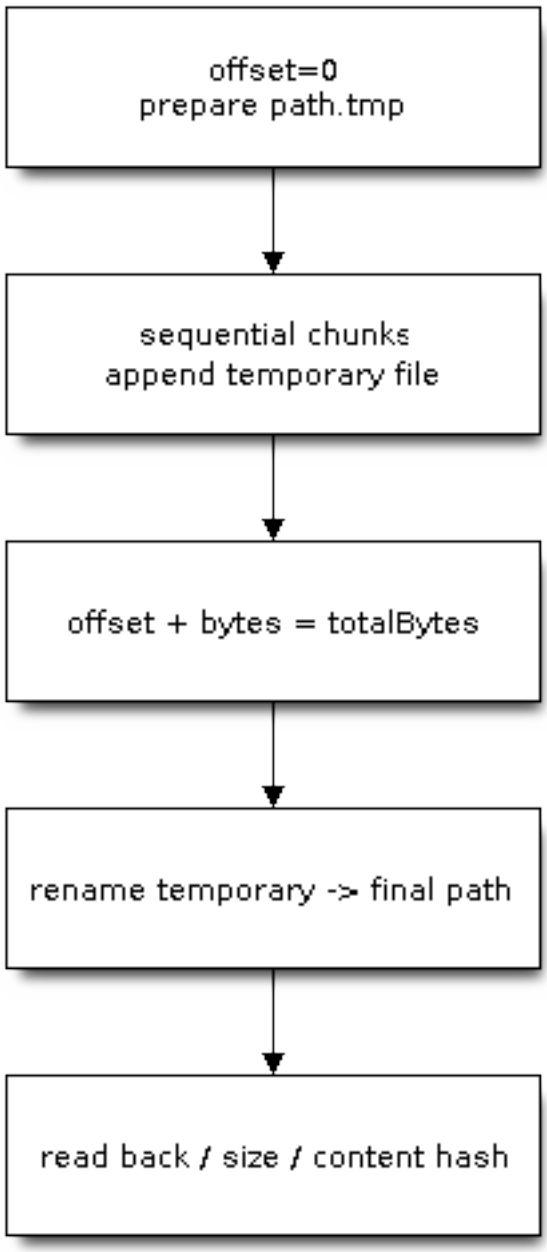


Figure 17: script.write uses the target .tmp file before the last chunk. The first chunk prepares it, later chunks

append, and the last chunk renames it. Do not report incomplete uploads as final available files.

The implementation **does not provide general random-offset writes or duplicate-chunk deduplication**. Nonzero offsets append rather than seek. Send sequentially, verify every ack bytes/offset/totalBytes/complete, and do not automatically replay a lost-response append chunk: it may duplicate content.

`complete:true` confirms the publication path completed, without proving a cryptographic whole-file hash. Verify important files by readback or supported device hashing. `runAfterWrite` only attempts to queue execution after completion; it proves neither startup nor successful execution.

Paths are limited by UTF-8 bytes. Current `MICROPY_ALLOC_PATH_MAX` plus the `.tmp` suffix gives a public script-path budget of 123 bytes, also subject to allowed-character validation. Both host preflight and final firmware checks remain necessary.

Advanced framed REPL helper and public reads

The advanced `EvRepl.writeFile()` helper (not `device.files.write()`) writes 224 raw bytes per chunk, hex-encoding them into Python input that must fit stdin. It writes `.viper.tmp` by default, closes it, removes the old target and renames; direct mode writes the destination. This client policy differs from RPC `<path>.tmp` and is not a general power-loss-atomic transaction.

Do not run parallel uploads sharing temporary files on one REPL. After cancellation/disconnect, verify file state and clean target handles/staging residues; clearing host promises alone is insufficient.

Reads request roughly 1 KiB raw data per step. The target returns hex text, actual total size and chunk byte count. Validate stable size, matching hex/chunk lengths, premature EOF and total deadline before building the result from actual bytes. A short output burst is not EOF.

The public `device.files.read()` total deadline defaults to 30 seconds. IDE may use bounded 60 seconds for verified large-file/TCP scheduling needs. Each subrequest still needs deadlines/cancellation; extending the overall budget does not authorize unbounded waits or replay.

Script execution is separate from upload ACK

```
{"seq":30,"channel":"user.rpc","type":"req","method":"script.run","encoding":"json
↪","sessionId":"89abcdef0123456789abcdef01234567","vmEpoch":1}

{"path":"/hello.py"}
```

```
{"ok":true,"path":"/hello.py","queued":true}
```

This reply only confirms `execfile(...)` was queued in REPL input, not execution completion. Imports, models, cameras and application code can fail or run indefinitely. Continue reading until an application marker, actual preview or explicit shared execution-lifecycle result.

Applications use `device.python.execute({ path, completion: 'start' })` for interactive runs and receive a handle with a `done` Promise. Core delegates to the session implementation internally; applications do not call `startScript()` directly. Ordinary infinite applications should not inherit bounded test-run automatic stopping. Run/Stop UI follows shared execution state rather than replacing internal REPL completion callbacks.

Bounded tests use deadlines, cancellation and confirmed stopping; interactive applications keep running with reader/keepalive active. A Run during upload must be rejected/queued by the coordinator, not interleaved as another script command.

Session lease and storage writer lease differ

USB MSC host writes and MicroPython VFS share one volume. EV-MUX ownership alone does not authorize safe concurrent writes. `device.control` actions `storage.writer.status` / `storage.writer.acquire` / `storage.writer.release` provide separate volume write ownership.

They involve volume (flash/sd), operationId, writer leaseId, current session and states including available/held/handoff_ready/quarantined. Handoff closes new I/O admission and drains outstanding operations/handles; failures can quarantine the volume. Query state and perform supported safe handoffs; CDC/MSC coexistence does not provide separate filesystem copies.

Session tokens, writer leaseIds and network operationIds may share formats but cannot substitute for one another. Follow capabilities and actual replies for writer transaction structure; do not assume file RPCs automatically acquire write permission.

Filesystem readiness is independent of writer ownership. Status replies also include `fs_state`, `fs_error`, `fatfs_result` and `format_supported`; an unmounted volume reports `writable: false`. Automatic initialization requires the corresponding board policy and exact `FR_NO_FILESYSTEM`; the default is enabled for internal Flash and disabled for SD. Disable the relevant policy to preserve unrecognized data. Other failures require diagnosis/user-directed recovery, and `storage.retry()` never formats. See *Storage, Filesystems and Data Protection* for media, backup and recovery policy and *storage –Diagnostics and Confirmed Recovery* for the explicit REPL API.

9.2.7 Preview, debug capture, backpressure and statistics

Applications explicitly publish preview frames

After application `img.flush()`, the preview producer encodes JPEG when delivery is needed and sends `preview.frame` data to the current user-stream owner. Read-only TCP observers may subscribe independently as described below. Bound USB webcam output is another independent consumer; see *uvc —USB webcam output*. It does not acquire an EV-MUX control session or initialize the camera.

UVC uses isochronous USB transfers. A failed or short payload aborts that frame: firmware does not append later JPEG payloads to its incomplete prefix. It attempts one header-only UVC `ERR|EOF` terminator, reports failure separately from complete-frame transmission, and releases the source slot. The next frame starts with a new FID and clear error flags. Bus reset or alternate-setting cancellation also releases the old source. This cannot guarantee lossless USB delivery; acceptance must decode every retained UVC frame, including concurrent CDC/TCP operation, rather than check only that the output file is nonempty.

```
import sensor
import time

sensor.reset()
try:
    sensor.set_pixformat(sensor.RGB565)
    sensor.set_framesize(sensor.QQVGA)
    while True:
        img = sensor.snapshot()
        img.flush()
        time.sleep_ms(200)
finally:
    sensor.shutdown()
```

The host explicitly interrupts the infinite loop; `finally` supplies resource cleanup. This device example guarantees neither delivery nor frame rate. Hosts must verify complete JPEG and session/epoch.

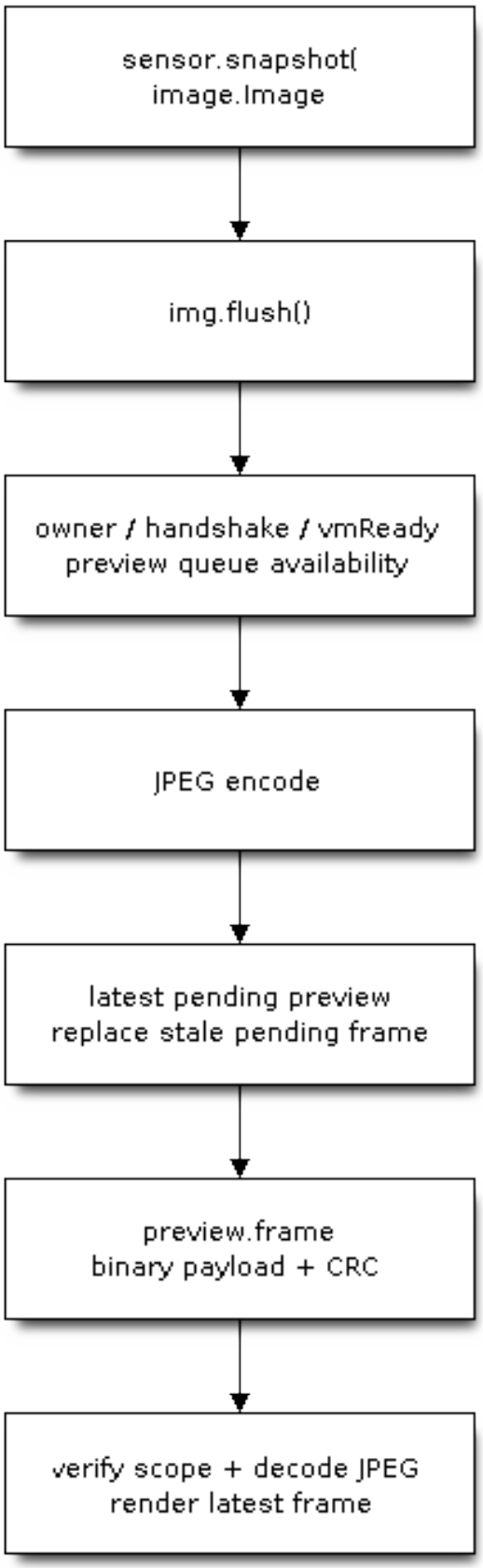

sensor.snapshot(
image.Image
img.flush()
owner / handshake / vmReady
preview queue availability
JPEG encode
latest pending preview
replace stale pending frame
preview.frame
binary payload + CRC
verify scope + decode JPEG
render latest frame

Figure 18: Check delivery eligibility and queue conditions before encoding/submitting images. Without an owner, observer or bound UVC consumer, preview is not encoded; continued camera acquisition remains application behavior.

An image frame

Conceptual metadata follows; actual seq, route, timestamp, sessionId and vmEpoch come from the current TX snapshot:

```
{
  "sid": "preview", "seq": 40, "channel": "preview.frame",
  "type": "data", "encoding": "binary", "route": "usb",
  "contentType": "image/jpeg", "width": 160, "height": 120,
  "ts_ms": 123456,
  "sessionId": "89abcdef0123456789abcdef01234567", "vmEpoch": 1
}
```

Payload begins directly with JPEG binary bytes. p is encoded JPEG size, not width times height or source RGB565 buffer size. Do not send JPEG payload to JSON/UTF-8 parsers.

`seq` helps observe frame order, not request retransmission of missing preview frames. Render the latest complete image; discard other-session/old-epoch images to prevent showing the previous program after soft reset.

Preview, camera capture and LCD screenshots

Table 16: Three explicit image sources

Property	`preview.frame`	`debug.capture_frame`	`debug.capture_screen`
Trigger	Application `img.flush()`.	Host camera-capture request.	Host LCD-screenshot request.
Source	Application-submitted image.	Last camera acquisition published by the application.	Last RGB565 frame submitted to display.
Wire type	Independent `type:data` JPEG.	Original RPC `type:rsp` carries binary JPEG.	Original RPC `type:rsp` carries binary JPEG.
Execution	Application owns an image and submits preview.	VM queue reads an independent immutable acquisition copy.	VM queue reads the display buffer and encodes without camera calls.
Delivery	Lossy, newest pending frame preferred.	Correlated by request seq/channel/method, session and epoch.	Same RPC correlation, or an error if no image is available.

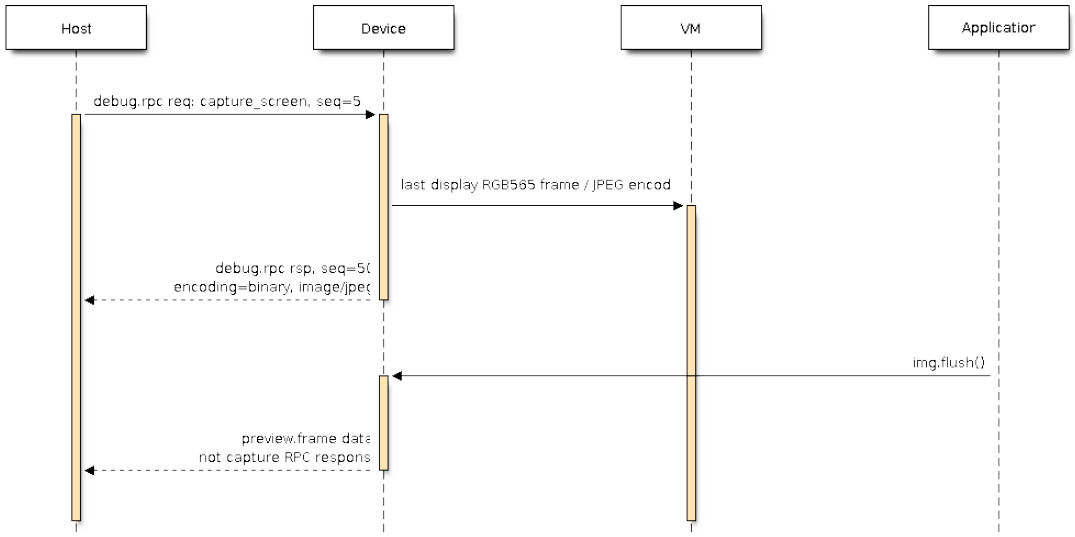


Figure 19: One-shot capture is delivered by its original RPC response, without waiting for another image channel. Core exposes `capture({ source: 'camera' | 'screen', format: 'jpeg' })` and never substitutes one source for the other. Successful results contain `{ data, mimeType, metadata }` with dimensions in metadata. Failures may be JSON errors; check encoding/contentType before decoding.

Before transmission, the shared JPEG encoder checks that its reported output length is positive and fits the allocated buffer, and validates the complete image structure against the source dimensions. This applies to camera capture, LCD capture and preview. Invalid encoder output is never queued as an image; capture returns `JPEG_FAILED`. Core validates JPEG structure and metadata dimensions after EV-MUX length, EOF and CRC checks and session/epoch filtering. An invalid response rejects only its matching capture with `INVALID_IMAGE`; an invalid preview is discarded with a diagnostic and never reaches image callbacks.

The image checks cover self-contained 8-bit Huffman JPEG headers, bounded marker segments and table references, scan boundaries and final EOI without trailing bytes. They do not decode entropy-coded pixels. Full decoder verification remains part of hardware acceptance. A disconnected link can still leave partial bytes on the wire; the receiver must wait for a complete validated EV-MUX frame and discard incomplete input, rather than display it.

Camera capture is passive and requires the advertised `debug.capture_frame.observed.v1` feature. It encodes the independent copy published after the application's last successful camera acquisition, including `sensor.snapshot(buffer)` with caller-provided storage and frames acquired by `sensor.skip_frames()`. It never initializes the sensor, takes another physical frame, writes the application's `Image` or follows a caller buffer after that acquisition returns. Later application drawing is visible through `img.flush()` preview or the LCD buffer, not through this raw-acquisition source. Device Core 0.6 refuses the unsafe camera RPC on firmware missing this feature; upgrading Core alone does not add safe firmware capture.

No valid publication returns `CAMERA_FRAME_NOT_READY`: before the first application acquisition, after an acquisition failure, or after camera reconfiguration/shutdown/reset or VM soft reset. An idle camera is not started to satisfy the request. A real change to frame dimensions, pixel format, mirror or flip invalidates the prior frame before changing hardware; a failed change may have altered the device and also leaves that frame invalid. Requests that leave these settings unchanged retain the current frame. An actual IPA mode change also invalidates before the controller stops/restarts capture, including a failed restart; an identical mode retains the frame. Manual ISP writes conservatively invalidate before accessing hardware, including partial failures, even if the requested values are unchanged. These changes also release any observation allocation-failure latch. The next successful application acquisition publishes the new dimensions/format. Camera observation and LCD capture never substitute for each other.

Camera publication, camera state mutations and JPEG encoding run on the main VM task. Python worker-task mutations are rejected with `OSError` instead of racing acquisition, encoding or cleanup. The transport task only queues requests; it never holds camera pixels. A queued request may expire with `VM_BUSY` / `VM_TIMEOUT`; cancellation, owner loss or an epoch change can discard its response without taking a frame or changing application storage. Soft reset stops request admission before it releases both camera stores. A hardware reset recreates them empty.

LCD capture does not initialize the screen, redraw it or start the camera. It reads only the last valid RGB565 display

frame. An absent LCD returns `NOT_SUPPORTED`; an uninitialized display or absent valid frame returns `DISPLAY_NOT_READY`. Reading and encoding run in the VM task, serialized with display operations and soft-reset cleanup. Busy VM work may return `VM_BUSY` / `VM_TIMEOUT`. Optional JPEG `quality` is 1–100. The result represents software display state rather than physical panel readback; it cannot prove backlight or panel hardware operation.

Use `--screen-output` on the unified Core hardware CLI to display a test pattern and save a JPEG. See *Communication acceptance, closed-loop tests and troubleshooting* for commands and acceptance criteria.

Queue priorities and dropped frames

Control, session, REPL, log and preview use distinct TX classes. Ordinary small-frame budgets are bounded and control has reserved capacity. A non-reading host cannot establish that all output is lossless.

Preview retains only the newest pending frame, replacing older unsent images. Already-transmitted bytes cannot be withdrawn or combine two JPEG tails. Preview and large debug responses share at most three demand-allocated 256 KiB buffers; this neither holds three arbitrary images nor permits 256 KiB host commands.



Figure 20: Limiting stale-preview latency takes priority over delivering every image. Saving all images requires device storage or a dedicated recording interface with separately designed capacity, playback and acceptance. Preview supplies no such guarantee.

UART deadlines depend on baud rate and total frame bytes; USB/TCP also use bounded waits. Writes may accept partial bytes or return WOULD_BLOCK; TX must retain the unsent suffix and continue within deadline. TX/socket failures can close TCP links; an RX partial-frame timeout only resets the parser of an otherwise live connection and does not imply device reset.

Log channels and system-log UI

IDE can combine `log.idf`, RPC diagnostics and connection events into a system-log panel, but that panel is not a ninth firmware channel. `repl.stdout` primarily displays application/REPL text; source mappings are in *Architecture, Execution Contexts and Standard I/O*.

Hosts may choose formatting, colors and filters without changing firmware ownership/routing. Identify unowned default/fallback logs, raw startup console and current-session application output separately in diagnostics.

Interpreting counters

`debug.info` scope `transport.stats` and `transport.state` remain queryable while Python is busy, exposing queues, RX rejections, TX timeouts, preview drops and backend state. Distinguish endpoint/link counters, global counters and application completion:

- Unread default USJ output can raise global drops without proving a particular active TCP RPC failed.
- A complete correlated RPC reply does not prove whole-application completion.
- Preview drops are permitted realtime behavior and do not automatically invalidate file transfer; accepting wrong-owner/epoch images is a protocol error.
- Reset/generation changes can alter counter scope; record sampling time, endpoint and session identity.

Keep diagnostics bounded to necessary counters/state. Do not flood the same TX queues with printing from failure paths.

Frame storage lifetime

LCD storage is allocated when the display is initialized and released at deinitialization. SPI color transfers finish before a draw returns, so later drawing cannot overwrite a buffer still borrowed by DMA. The application's default camera snapshot buffer is allocated on its first use; successive application captures may reuse that buffer, while an explicit caller buffer belongs to the application. Python must still copy an image it wants to retain across its own next capture or camera reconfiguration.

Safe camera observation owns a separate raw-frame copy, allocated after the first successful application acquisition and reused for stable geometry. For example, QVGA RGB565 adds 153,600 bytes to the application snapshot storage; it does not reserve memory at boot and does not allocate a new buffer per host request. The copy prefers PSRAM and falls back to an 8-bit-capable internal heap; configuration alone does not prove PSRAM initialized. Only CPU-owned output is copied: board backends finish their DMA/PPA operations and cache synchronization before publishing. No observation buffer is submitted directly to camera DMA.

A failure to allocate the observation copy preserves application capture success and pixels. Camera observation then returns `NO_MEM` until this allocation-failure state is released, even if a later individual acquisition fails; allocation is not retried on each subsequent frame. A camera reconfiguration/shutdown/reset or VM soft reset releases this state so a later application acquisition can retry. Actual reconfiguration releases the observed copy before touching hardware, and VM cleanup also releases the default application buffer. A borrowed firmware observation view remains valid only during that VM call; Core returns independently received JPEG bytes on the host. Driver DMA, PPA, JPEG and transport buffers have separate lifetimes and must be included in the total image-memory budget.

Read-only TCP Preview

A preview client may connect to the same TCP port (3232 by default) and send `user.rpc/preview.subscribe` with payload `{}` within the 3000 ms handshake deadline. It must not acquire a control session first. The response is `{"ok":true,"subscribed":true,"channel":"preview.frame","maxObservers":2}`. `preview.unsubscribe` with `{}` stops publication on that connection. Both operations are idempotent and do not require `sessionId` or `vmEpoch`. Capabilities advertise `preview.observers.v1`, `preview.subscribe` and `preview.unsubscribe`.

Up to two observers can watch alongside the exclusive USB, USJ, UART or TCP controller. A third observer receives `OBSERVER_LIMIT`; one of the three TCP slots remains available for control. An observer's `hello` returns `READ_ONLY`; Python, files, REPL and device control remain subject to owner authorization. To become a controller, use a separate connection. Observers receive only the explicitly published image channel and replies to their own requests; preview metadata does not expose the controller's session ID or execution identity. Its `route` field retains the current user-stream route; the observer socket is independent. This role separation does not add network authentication.

Applications publish processed images with `img.flush()`. Connecting does not start sensor capture. The existing JPEG preparation is shared with the control preview and bound UVC output; no HTTP server or additional JPEG encoding is introduced. Observers share one immutable JPEG copy and one CRC calculation per publication, with independent TX cursors. TCP coalesces the frame header, metadata, JPEG and trailer with scatter/gather `sendmsg` rather than issuing a separate send for every segment. It still sends a separate byte stream to each observer, so bandwidth grows with the number of viewers. The existing owner preview pool remains separate.

RX parsers are allocated statically for real endpoint slots rather than a rectangular transport/slot table, avoiding unused serial parsers as TCP capacity grows. Temporary TCP send memory exhaustion is treated as backpressure within the same bounded deadline. Socket close runs outside the shared I/O lock, with a short send timeout bounding lwIP FIN-allocation retries. `debug.info` with `scope:transport.stats` includes `lastTxTimeout` (transport, slot, byte offset, total bytes, elapsed milliseconds and idle milliseconds) for diagnosis.

At most three lazily allocated, reusable 256 KiB PSRAM slots serve observers (768 KiB maximum), with two small frame descriptors per observer. Frames that have not started may be replaced by the latest publication. Read-only frames have an 8000 ms absolute deadline and a 4000 ms no-progress deadline, advertised as `tcp.observerFrameTimeoutMs` and `tcp.observerIdleTimeoutMs`. Progress refreshes the idle timer; replacing a frame before its first byte does not refresh either timer. Started frames finish within those budgets or their socket is closed; a slow observer does not hold the controller or another observer's queue. Generation checks prevent

old frames or subscriptions from reaching a reconnected socket. Subscribed sockets need no control lease or keepalive and remain subscribed while idle; `preview.unsubscribe` returns the socket to normal idle reclamation.

The host helper uses Python's standard library and its bounded EV-MUX parser. `--view` additionally requires `ffplay` on PATH; its bounded display queue keeps only the latest pending image. The wire parser retains frame boundaries and validates JPEG CRCs.

```
python tools/preview_tcp.py 192.168.1.100 --view
python tools/preview_tcp.py 192.168.1.100 --duration 10 --save-last preview.jpg
```

Read-only acceptance can run explicitly with `tools/test_preview_tcp.py --host <device-ip> --port <serial-port> --device-cli <installed-plugin>/cli/device-cli.mjs --output-dir <project-local-directory>`. Add `--video <V4L2-node>` to check concurrent UVC. This test replaces the running Python application with bounded test programs, uses Pillow to decode received JPEGs, and restores the UVC logo afterward.

9.2.8 Host integration, Wi-Fi configuration and recovery

First-time Wi-Fi setup and wireless connection

Connect a Wi-Fi-capable board running compatible ESP-VISION firmware to your computer with a USB data cable. The computer and board need a mutually reachable LAN. See *LAN discovery protocol and board defaults* for board support and *Build vision applications with AI* for first-time firmware installation.

Set up Wi-Fi on the website You can also configure Wi-Fi directly on the ESP-VISION website without installing local host tools.

1. Connect the board over USB and leave ESP-VISION firmware running normally. Open the website in a desktop browser supporting Web Serial, such as Chrome or Edge, and find the browser flashing panel.
2. Under "Flash Mode", choose "Configure Wi-Fi Only", click "Connect Serial", and select the board's running serial port in the browser dialog.
3. Under "Device Wi-Fi setup", enter the Wi-Fi SSID and password, then click "Save".
4. Wait for "Wi-Fi Connected" and the TCP address. The configuration is saved on the board for automatic reconnection after power-on. To check again, click "Read Status Again".
5. Click "Disconnect" to release the website's device connection, then connect your AI assistant or another host tool to the board over the LAN.

"Configure Wi-Fi Only" updates network settings without reinstalling firmware. Use it directly when compatible firmware is already installed; after first-time firmware installation, you can also complete setup on the website's follow-up Wi-Fi page.

Set up Wi-Fi with an AI assistant After installing the ESP-VISION plugin, describe what you want:

Note: Connect my board to Wi-Fi so I can update its program and view the camera over a wireless connection.

---

1. Keep USB connected, give the assistant the Wi-Fi name, and follow the password-entry prompt.
2. Wait for the assistant to confirm setup and report the board's LAN address. Initial setup needs time to obtain an IP address and save the configuration.
3. Continue with an application request over the wireless connection, such as "Track red objects with the camera and show me the result on my computer."
4. Once the wireless connection works, you can use a separate power supply suitable for the board. On subsequent boots, the board reconnects using its saved configuration.

Set up Wi-Fi with the CLI For manual setup, install Device CLI and its serial dependencies as described in *Device management CLI*. In the commands below, replace the example values `COM3`, `MyWiFi` and `192.168.1.100` with the discovered serial port, your network name and the IP address reported by the board.

1. Find the board connected over USB:

```
esp-vision-device inspect --interface serial
```

2. Configure Wi-Fi and enable the network service; enter the password at the terminal prompt, which hides your input:

```
esp-vision-device configure --port COM3 --ssid "MyWiFi" --network-enabled true␣
↪--timeout 60000
```

   The command waits for the network configuration transaction to finish, with the wait budget set to 60 seconds here. It returns `ok: true` on success; setup succeeds only after obtaining an IP address and saving the configuration. For non-interactive input, see `--password-stdin` in the CLI reference.
3. Check the board's network status and address over USB:

```
esp-vision-device inspect --port COM3 --fields identity,network
```

   Check that `network` reports `enabled` and `configured` as `true` and `state` as `online`, then note `ipv4`. If a command times out or fails, inspect this status and errors in `operation` / `lastOperation` before deciding whether to configure again.
4. Discover LAN devices and verify wireless access using the reported IP address:

```
esp-vision-device inspect --interface tcp
esp-vision-device inspect --tcp 192.168.1.100:3232 --fields identity,network
```

   Compare identity with the USB result to confirm it is the same board. Once connected, CLI file management, Python script execution and image capture can use `--tcp`; see *Preview, debug capture, backpressure and statistics* for continuous image preview. Keep the board powered during wireless development; firmware flashing still uses USB serial.

If setup fails or the board is not discovered

- Incorrect password or network not found: check the Wi-Fi name, password, signal and the board's supported frequency bands, then configure again over USB.
- Connected but unable to obtain an IP address: check the router's DHCP service and inspect network status over USB.
- The board has an IP address but is missing from discovery: try connecting directly to that IP; check LAN reachability, guest-network isolation, firewall rules and mDNS filtering.
- Address changes or connection drops: discover the board again; if it remains unreachable, keep USB power and connectivity and use the network status command above to investigate.

Recommended host stack

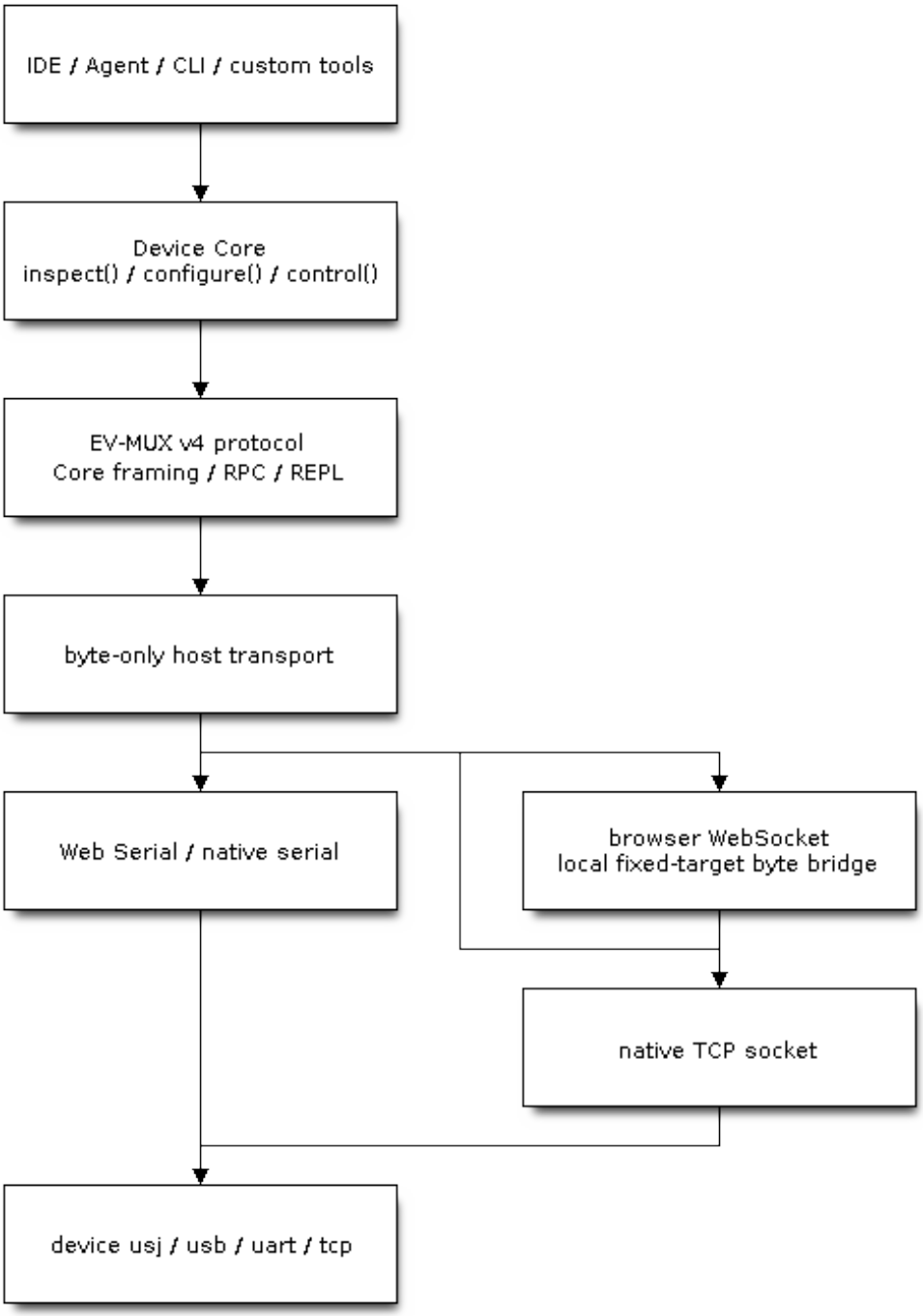


Figure 21: Shared Core owns protocol state; transport carries bytes. A network bridge must not acquire again, re-encode frames or execute Python, creating duplicate owners/correlation logic.

`@esp-vision/device-core` in `host/device-core` provides the shared protocol implementation. IDE, independent Agent and firmware hardware checks all use Core; Agent workflows, model providers and MCP plugins belong to the Agent host, not Core or firmware communication. Applications obtain the public API through `createDevice()`. Parser, framed REPL and session keepalive remain internal rather than being assembled again by each host.

See *Device Core integration guide* for complete Node TCP, browser Web Serial and high-level API examples. Hardware testing uses the package's single `npm run test:hardware` entry point. Its serial adapter only forwards bytes; it is not a second Python EV-MUX client. See *Communication acceptance, closed-loop tests and troubleshooting* for camera and LCD check selection.

This adapter detail expands the shared *Solution Architecture* diagram: host applications → Device Core → EV-MUX → device firmware. Core owns protocol/session state; native serial, Web Serial, TCP and WebSocket bridges only move bytes. DNS-SD discovery precedes connection and ROM flashing is a separate loader path, not EV-MUX image transfer.

Integration sequence

1. Select a physical endpoint, verify device/USB topology and open transport.
2. Serial performs one bounded startup drain before its first request; TCP acquires immediately.
3. Start the continuous reader, acquire through hello with a new nonce, validate evMuxVersion/correlation and retain session.
4. Query capabilities/firmware identity and negotiate command/REPL budgets/timeouts.
5. Start reported-interval keepalive; keep the reader active throughout application operations.
6. Wait for current-epoch vmReady and verify executable REPL before VM work.
7. Coordinate files, run, stop and reset serially while receiving logs/preview.
8. Release session when possible before closing transport, reader and pending requests.

Why serial startup drain runs only once

Opening USB serial may reveal old complete/partial frames, tracebacks and IDF startup text; it does not trigger a discovery hello. `serial.flush()` normally waits for host TX rather than reading/clearing device replies.

At connection startup Core receives and processes startup bytes through the same parser for a target 300 ms; this does not establish a business session. If a possible incomplete header/body crosses that window, Core continues receiving within the parser's remaining progress and total frame budgets (both default to 12 seconds). It keeps rejecting startup frames and sends no hello until parsing reaches a complete frame boundary. The total deadline is fixed when this extra wait begins; successive frames cannot extend it. Cancellation, disconnection or an expired receive budget fails the connection and cleans up. A parser timeout that discards an incomplete frame is a failure, not a completed boundary; Core never flushes or truncates buffered bytes to manufacture success.

Drain reduces observed old data; it neither proves every device/USB/host buffer empty nor diagnoses USB completion failures. Never clear input before every subsequent request, losing keepalive/preview/replies.

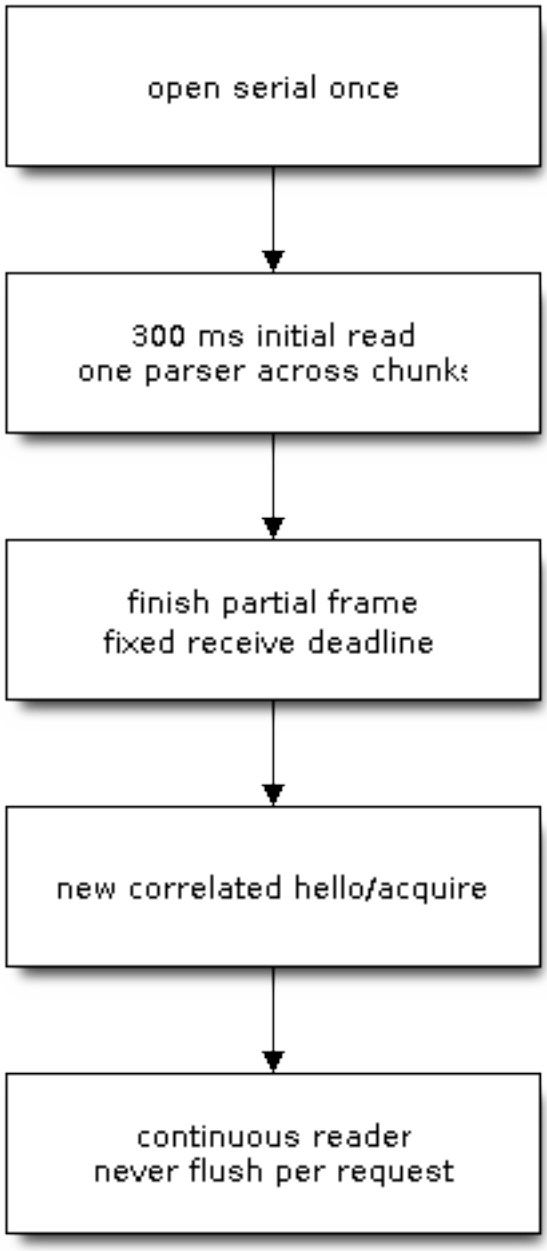


Figure 22: Drain prepares integration; it does not replace completion correlation. Continue rejecting stale seq/session/epoch.

One hardware-test entry point

Run from the repository root with Node.js 22. Serial mode uses npm native bindings; neither serial nor TCP requires host Python. First verify identity and camera on an online device, then run all checks on USB:

```
npm --prefix host/device-core run test:hardware -- --tcp <device-ipv4>:3232 --
↪checks identity,camera
npm --prefix host/device-core run test:hardware -- --serial <selected-port> --role␣
↪usj --checks all --timeout 600000
```

Checks stop existing Python programs and may initialize the camera or display a test pattern. Schedule exclusive device use and close the owning IDE/terminal first; switching interfaces does not bypass a valid owner. See *Communication acceptance, closed-loop tests and troubleshooting* for checks, LCD JPEG output and acceptance evidence.

Actual Wi-Fi/NVS state machine

Initial configuration requires an acquired connection, usually USB/USJ/UART. An unconfigured TCP server cannot provide entirely wireless first-time provisioning. Native/Hosted Wi-Fi and Python WLAN share the underlying instance; background service creates no second WLAN driver.

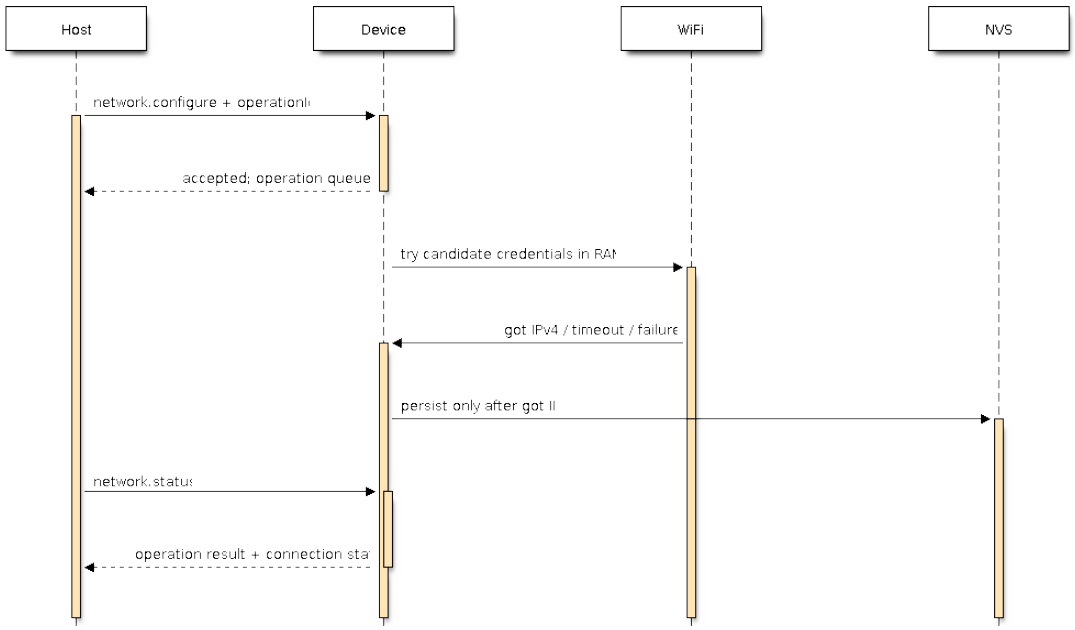


Figure 23: Acceptance, connection and persistence need separate confirmation. Changes start after accepted-reply TX, waiting at most about 1000 ms. Configure tries the candidate in RAM, persists only after obtaining IP within approximately 30 seconds, and restores saved configuration on failure.

```
{
  "action": "network.configure",
  "operationId": "fedcba9876543210fedcba9876543210",
  "ssid": "<wifi-ssid>",
  "password": "<wifi-password>"
}
```

Send through `user.rpc/device.control` with current sessionId in metadata. Network control does not require idle VM. `network.status` queries final state, `network.enable` controls persisted enablement and `network.forget` persists forgotten configuration; these are not Python WLAN method names pasted into RPC.

Use a fresh 32-character operationId per mutation, reusing it only to recover the same transaction with identical payload. `network.changed` hints at change; status is a bounded correlated query. Never infer latest transaction state from old events after reset/disconnect.

Existing NVS records take priority, including disabled, forgotten-empty and corrupt records; factory macros do not override them. Factory defaults persist then attempt connection only when no record exists. RPC user configuration obtains IP before saving: distinct source flows.

Python WLAN mutations return EBUSY during background transactions. Ordinary Python changes may temporarily affect runtime without implicitly rewriting persistent service records. Disabling background service still allows explicit Python WLAN use; TCP service and policy follow service lifecycle.

Examples use placeholder credentials; redact host logs/diagnostics. This TCP implementation has no authentication/encryption and follows a trusted development-network boundary.

Browser network integration

Browsers cannot open arbitrary TCP sockets. IDE uses a local fixed-target WebSocket-to-TCP binary bridge with explicit origins, a fixed device address, one active browser and TCP closure on browser disconnect. The bridge neither interprets EV-MUX nor acquires sessions.

The bridge belongs to the separate IDE repository; do not assume its npm scripts exist in the firmware root:

```
npm run network:bridge -- --device <device-ipv4>
```

Local IDE typically uses `http://127.0.0.1:8766/` and `ws://127.0.0.1:8766/device`. Remote HTTPS needs the exact permitted origin and browser local-network/security permissions; use a trusted WSS proxy when TLS is needed. USB Web Serial and network bridges use the same `createDevice` API and byte-transport contract.

Confirming reset and cancellation

Soft reset: wait for the same owner with new vmEpoch, vmReady and REPL barrier. Asynchronous event/hello has a separate observation budget; absence alone is not reset failure, and presence alone is not executable REPL.

Hard reset/ROM: invalidate sessions/requests, await enumeration/runtime, reacquire and verify firmware identity. Record flashing verification/reset states separately; full readback verified without observed runtime restoration does not justify automatically rewriting images.

Cancellation: stop further upload/execution, interrupt target work where possible and confirm state. Preserve unknown completion where necessary. Recover network transactions by operationId; ordinary RPCs, REPL input and file append have no universal replay guarantee.

Host timeout budgets must remain distinct

- Discovery/capabilities: reference correlated RPC default about 6 seconds.
- Keepalive: about 3 seconds per request, within firmware total lease.
- Serial startup drain: target 300 ms, once on initial connection; a crossing partial frame may finish within its remaining receive budgets, without extending the fixed total deadline.
- Host RX frame assembly: reference budget about 12 seconds, allowing multi-second TCP read gaps.
- Device RX: approximately 3/10 seconds idle/frame; opposite direction from host large-image reception.
- VM queue: expires around 5 seconds; longer host waits do not enlarge it.
- File readFile: Core total default 30 seconds; tested IDE scenarios use bounded 60 seconds.
- Wi-Fi configure: attempt about 30 seconds; accepted only starts it, requiring final-state queries.

These are current defaults/reference values. Use actual capabilities and client options. Retry within bounds only for explicitly idempotent recovery; slow networking does not justify indefinite waits or automatic duplicate application commands.

9.2.9 LAN discovery protocol and board defaults

Note: The implementation advertises an ESP-VISION DNS-SD service and provides Device Core discovery adapters for Node and the browser bridge. Firmware build/hardware qualification remains a separate release gate. Existing hello discovery operates only after a transport is connected.

Lightweight protocol selection

Use mDNS/DNS-SD for the first implementation. It needs no central server and reuses the existing `espressif/mdns: ~1.1.0` dependency shared by IDF 5.5, 6.0 and 6.1 manifests. DNS-SD uses PTR for enumeration, SRV for host/port, TXT for metadata and A records for the current IPv4 address.

Table 17: Implementation tradeoffs

Protocol	Cost and benefit	Decision
mDNS/DNS-SD	Existing component and host tooling; caches and naming conflict handling are standardized.	Preferred; register one service with bounded TXT metadata.
Custom UDP query/reply	A smaller dedicated responder is possible, but requires a new port, parser, retry/cache and multi-interface rules.	Reserve for a measured resource constraint; do not ship two discovery protocols initially.
SSDP/UPnP	Adds another discovery format and, for full UPnP integration, device descriptions.	No advantage for this existing EV-MUX endpoint.

The pinned mDNS 1.1.0 Kconfig defaults to a 4096-byte task stack. This is not total heap usage. Reuse its single existing instance rather than creating another discovery task/socket. Measure image size, free/minimum internal heap and stack watermark before and after service registration; no measured RAM delta is available yet. Avoid continuous application-level beacons; use the component query, announcement and TTL behavior.

Service contract

- Browse `_esp-vision._tcp.local` on the local link using mDNS UDP port 5353.
- Hostname: `esp-vision-<full-board-name>-<last-6-mac-hex>-<firmware-version>.local`; normalize board/version to lowercase DNS labels with hyphens. Version is the firmware build version, never the IDE version. Names stay within 63 bytes; long versions retain a prefix plus a deterministic eight-digit build digest, while TXT retains the original version. Main-chip MAC is used on every board. The hostname label (without `.local`) and full device ID are identical in hello events/responses, capabilities and device debug information over USB, USJ, UART and TCP.
- Instance display name: board name plus a short identity suffix. SRV advertises the actual `ESP_VISION_TCP_PORT` (default 3232); do not duplicate the port in TXT.
- TXT keys: `txtvers=1`, `id=<device-id>`, `board=<board>`, `fw=<firmware-version>`, `ev-mux=4`, `hostname=<canonical-hostname>`. Keep the total encoded TXT payload within 256 bytes and validate lengths before registration.
- Reuse the Launcher device identity derived from the main chip eFuse MAC, including on Hosted Wi-Fi boards. Do not substitute the companion radio MAC, IP, boot ID or session ID. If stable identity is unavailable, report discovery unavailable instead of inventing a random permanent identity.
- Never advertise passwords, session credentials or a periodically refreshed owner state. Discovery metadata is untrusted; it is not authentication.

Firmware lifecycle and Python coexistence

The platform network service owns mDNS initialization and coordinates with `overlay/micropython/ports/esp32/network_wlan.c`, which currently initializes mDNS and sets the MicroPython hostname. Do not retain two independent initialization flags. Python mDNS queries reuse the shared instance. `network.hostname()` returns the same canonical name; assigning another hostname raises ValueError, preserving one firmware-defined identity. DHCP and the Python port use the same name. Soft reset does not regenerate it. ESP-IDF network-interface hostnames are aligned to the 63-byte DNS limit; DHCP option space is set to 255 bytes to accommodate that label alongside other options.

Register the service only after STA has IPv4 and the EV-MUX TCP listener has successfully bound. Withdraw it when the listener closes, Wi-Fi is disabled or STA disconnects; advertise the new address after reconnect. An unexpected power loss is handled by host TTL expiry. Soft reset must not duplicate the service or stop the platform

discovery responder. mDNS failure is observable but must not stop working TCP/USB access. The shared network sdkconfig limits mDNS to the STA interface, matching the TCP bind address; AP/Ethernet interfaces do not advertise this endpoint.

Host and Device Core integration

Device Core owns the discovery-result contract and cancellation/timeout semantics; an injected environment adapter performs network browsing. Node hosts can use a DNS-SD adapter directly. Browser IDE obtains results through the local Node bridge because ordinary web pages cannot browse multicast DNS. This does not introduce Agent dependencies into Core.

Results include stable device ID, display name, board, firmware, EV-MUX version, hostname, IPv4 addresses, TCP port and interface identity. Cache by stable ID while retaining interface/address candidates; remove expired addresses and retain valid alternatives. Hostname alone is not a device identity. Use bounded queries, support AbortSignal and release browse resources on cancellation.

Browsing does not connect to every TCP endpoint or acquire ownership. After user selection, Core connects and issues non-acquiring hello to verify v4, stable device identity and current occupancy, then explicitly acquires the session. Both acquiring and non-acquiring hello responses include stable `deviceId` and `hostname` fields. When `expectedDevice` is supplied to `DeviceSession.negotiate`, a non-acquiring hello verifies these fields before requesting ownership. Report an occupied session; takeover continues to follow the existing explicit policy.

The bridge only browses and forwards bytes; EV-MUX remains in Core. Discovery HTTP/control endpoints require the same exact-origin and loopback policy as the byte bridge. Select a resolved candidate through a bridge-issued opaque handle, revalidate its cached address/port, and retain an explicit fixed-target manual mode; do not expose unrestricted browser-selected TCP forwarding.

IDE places refresh and discovered-device selection in the existing network connection section. Keep manual IP/hostname input for networks with multicast filtering, client isolation or separate VLANs. Local Network Access permission does not itself provide DNS-SD enumeration.

Default-enabled Wi-Fi board policy

Wi-Fi-equipped boards must enable WLAN, `ESP_VISION_ENABLE_BACKGROUND_NETWORK=1` and `ESP_VISION_ENABLE_TCP_TRANSPORT=1` by default. The current seven Wi-Fi configurations already satisfy this policy: `ESP32_P4X_EYE` and `ESP32_P4X_FUNCTION_EV_BOARD` use Hosted Wi-Fi; `ESP32_S3_EYE`, `AtomS3R-M12`, `ESP32_S31_CHATBOT`, `ESP32_S31_KORVO` and `ESP32_S31_MOSAICO` use native Wi-Fi. The two Hosted boards include `sdkconfig.p4_wifi_common`.

`ESP32_P4X_VISION` currently disables WLAN and has no Hosted Wi-Fi configuration. P4 alone has no native Wi-Fi; confirm an attached radio and its board wiring before enabling this port. `TEMPLATE` also remains hardware-dependent. Do not copy Hosted SDIO/reset pins from another board.

Default-enabled means the firmware service is available at startup without a Python script. Empty factory SSID remains unconfigured, valid saved profiles reconnect, and saved disabled/forgotten intent is preserved. TCP starts listening only when network conditions permit. Discovery follows the TCP listener lifecycle and the same default policy.

Implementation and acceptance

1. Firmware: unify mDNS lifecycle, register/withdraw the service and expose stable identity in non-acquiring hello.
2. Core: define adapter/results, cancellation and selection-time identity checks; keep Node and browser discovery separate from the EV-MUX parser.
3. IDE: extend the local bridge and connection panel; retain manual addressing.
4. Verify multiple boards, occupied sessions without takeover, multiple host interfaces, DHCP address changes, disable/re-enable, abrupt power loss/TTL, soft reset, discovery cancellation and filtered multicast fallback. Run the full firmware matrix in CI and measure resource deltas on native and Hosted hardware.

Device Core discovery examples

Node installs the optional multicast-dns adapter dependency; browsers use the existing local bridge. Start the bridge with `npm run network:bridge` (no fixed IP required), then refresh/select a device in the IDE network panel. Candidate handles expire after at most 30 seconds; refresh before reconnecting an expired selection.

```
npm install @esp-vision/device-core multicast-dns
```

```
import { createNodeDiscovery } from '@esp-vision/device-core/discovery/node';
const devices = await createNodeDiscovery().list({ timeout: 3000 });
// Optional: createNodeDiscovery({ interfaceAddress: '<local-interface-ipv4>' })
```

```
import { createBridgeDiscovery } from '@esp-vision/device-core/discovery';
const devices = await createBridgeDiscovery('ws://127.0.0.1:8766/device').list();
// Connect the chosen connectionUrl, then:
// await session.negotiate({ role: 'tcp', expectedDevice: devices[0] });
```

Protocol references

- Multicast DNS, RFC 6762
- DNS-Based Service Discovery, RFC 6763
- Espressif mDNS 1.1.0 Kconfig

9.2.10 Device Core integration guide

Device Core documentation is published in this documentation center. See *Device Core host SDK* for the SDK overview, *Application API reference* for every application API, *Advanced export API reference* for advanced exports and *Integrating the Device Core npm Package* for runnable TCP/Web Serial adapters.

9.2.11 Communication acceptance, closed-loop tests and troubleshooting

Four levels of acceptance evidence

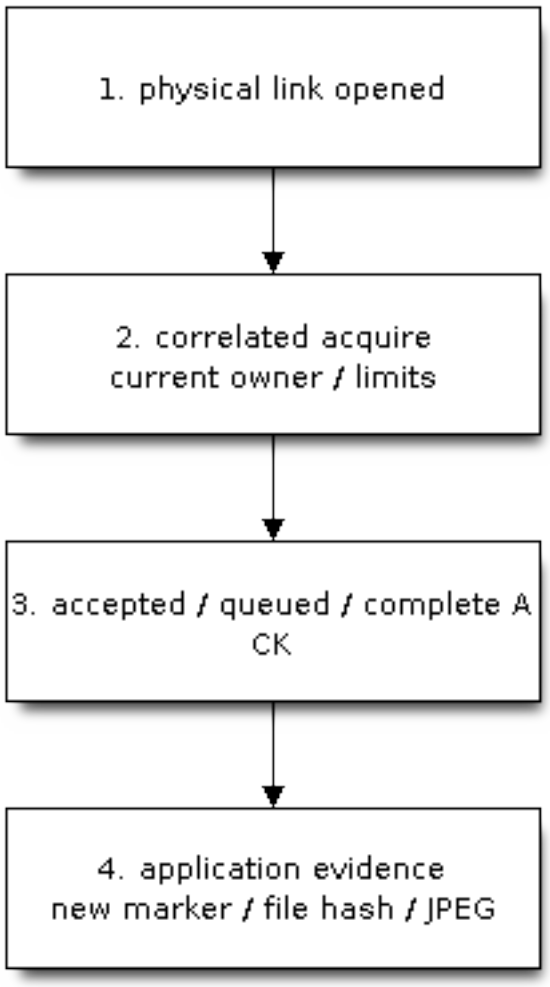


Figure 24: Lower-level success does not replace application evidence. Opening serial, acquiring ownership, upload ACK and an actual camera image establish four different conclusions.

A responsive control plane is insufficient to report files/scripts/preview or a complete hardware matrix passed. Record board/physical identity, installed firmware commit, endpoint, session/epoch, request stage, budgets and application evidence.

Suggested acceptance matrix

Table 18: Verify functions and boundaries

Category	Operation	Success evidence
framing	Fragmented metadata/payload, coalesced frames, bad CRC, SOF/EOF inside payload and UTF-8 across frames.	Only complete valid frames delivered, bounded resynchronization and no delimiter truncation.
Session	Two contenders, same-nonce retries, release/reacquire, lease expiry and TCP slot reuse.	One owner, correct errors and no old-link/session input/output on new connections.
VM	Busy scripts, Ctrl-C, soft reset, new-epoch readiness and delayed old output.	Actual new REPL marker; old epoch cannot complete new requests.
Files	Empty, Unicode/binary, chunked/64 KiB, cancellation and unknown publication.	Verify bytes/size/content hash and preserve originals/protected files.
preview	Continuous JPEG, slow readers, serial switching/reconnection and interleaved RPC.	Complete correctly sized/current-scope images; drops allowed, stale-owner data rejected.
Network	Configuration accepted/final IP, disable/restore, FIN/RST and silent clients.	Final transaction state, distinct owner/slot reclamation and recoverable new connections without replay.
Download/reset	Python/USB-control requests, ROM enumeration, write/readback and runtime acquisition.	Separately prove mode transition, image verification and new runtime identity, not mere port loss.

Record actually installed firmware, distinguishing source inference, CI compilation and tests on another board from this board acceptance. Browser bridge tests must state whether native serial chooser/DTR were covered.

Unified Core hardware checks

Protocol and application hardware tests use the single `npm run test:hardware` entry point in `host/device-core`, calling the public `createDevice()` API. Camera, file and LCD checks share Core rather than maintaining independent host protocol clients or an Agent smoke entry point. The serial adapter only carries bytes, while TCP uses Node sockets; no Agent installation is needed.

Use Node.js 22. Serial mode uses npm native bindings; no host Python is required. Keep all outputs inside the project directory. Run these commands from the repository root:

```
npm --prefix host/device-core test
npm --prefix host/device-core run check:schema
npm --prefix host/device-core run check:package
npm --prefix host/device-core run test:hardware -- --serial <USJ-port> --role usj -
↪-checks all --timeout 600000
npm --prefix host/device-core run test:hardware -- --serial <USB-OTG-port> --role␣
↪usb --checks identity,camera,display
npm --prefix host/device-core run test:hardware -- --tcp <device-ipv4>:3232 --
↪checks identity,memory,camera
```

`--checks` accepts `all` or comma-separated check names. Camera/display checks include `identity`, `memory`, `camera`, `framesizes`, `pixformats`, `orientation`, `ipa_stats`, `snapshot_rate`, `preview`, `image_ops`, `display`, `display_stream`, `camera_display_lifecycle`, `image_copy_to_fb`, `image_transform_argument`, `sensor_snapshot_buffer`, `memory_policy`, `espdl_inference`,

`isp_ccm_frames` and `isp_lsc_display`. `all` selects the full suite, not a promise that every board has the required hardware. Reports must distinguish passes, failures and unsupported-capability skips; a skip is not a pass. During experimentation test only the current hardware; run the full firmware matrix in CI for final submission.

ISP image and allocation checks Use a stationary illuminated target and select the exact board and installed firmware version for an ISP investigation:

```
npm --prefix host/device-core run test:hardware -- --serial <USB-OTG-port> --role␣
↪usb --checks isp_ccm_frames,isp_lsc_display --expect-board ESP32_P4X_EYE --
↪expect-firmware-version <installed-version> --frames 400 --require-all
```

`isp_ccm_frames` writes each controlled red/grey CCM matrix once and records eight consecutive rendered-frame samples with frame numbers, elapsed time, RGB sums and mean saturation. It deliberately includes a two-second sleep before one grey sequence: the first buffer may still be old, while continued consumption must reveal the new output within the bounded sequence. It also verifies that IPA rejects manual writes and that rewriting an identity CCM preserves explicit ISP white-balance gains in both readback and rendered RGB ratios. These ratios require a stationary target; they do not measure or freeze the sensor's internal automatic white balance. Fewer than sixteen sufficiently bright samples fails the check as unobservable, rather than treating a dark frame as proof that CCM works.

`isp_lsc_display` uses QVGA RGB565, the reported LSC gain-table size, the report's 1.0-to-0.5 quadratic ramp, and a centered camera image drawn onto a full-panel canvas before `lcd.write()`. After warming both LSC table banks and display staging, it runs at least eighty frames (`--frames` raises that count), writes CCM twice per frame, and toggles LSC every twenty frames. Reports include free and largest-block DMA/internal/PSRAM heaps, invalid-table rejection followed by a successful display write, resource release and a fresh camera/LCD lifetime. Heap history retains at most ten samples so observation storage stays bounded. A bounded heap tolerance allows small unrelated allocations; inspect the recorded series for trends during longer runs. This is software-frame and resource evidence. Compare the actual LCD while LSC switches to establish whether the physical panel is free of corruption; a passing check or software screenshot cannot establish that alone. A pass on current firmware does not retroactively identify the root cause of an older firmware report.

CI target-test contract The CI hardware gate runs eighteen explicitly selected runtime checks, including `esp-pdl_inference`, on the P4 Function EV Board with ESP-IDF v5.5.6. The two ISP checks remain separate manual acceptance on an illuminated stationary target; adding a registered check does not silently expand the unattended CI suite. CI retains both the camera JPEG and the LCD test-pattern JPEG, with dimensions validated against the live display. All selected checks must pass (`--require-all`); an unsupported-capability skip fails this board's gate. Camera-mode loops tolerate only explicit unsupported errors; allocation failures, timeouts and unexpected mode-switch errors fail the check. This proves the selected board/IDF pair, not the other boards in the compilation matrix. The driver passes `--expect-board` and the exact `project_version` from the downloaded build's `project_description.json` to both startup probes and the full suite, rejecting the wrong board or stale firmware before Python/camera/display tests.

To qualify one board through CI, set `ESP_VISION_TEST_BOARD` and `ESP_VISION_TEST_IDF_IMAGE` to its exact firmware matrix pair, and explicitly set `ESP_VISION_TEST_HARDWARE=1`. For P4 Function EV Board use `ESP32_P4X_FUNCTION_EV_BOARD` and `v5.5.6`. Only the matching firmware and hardware jobs are selected; without the hardware opt-in, a targeted run only builds firmware. `ESP_VISION_DEVICE_CORE_ONLY=1` still disables all firmware and hardware jobs.

Runtime fixtures load into an isolated RAM namespace using bounded `python.execute()` calls. Each encoded command fits the firmware's advertised `replInputMax` (at most 511 bytes); a larger request would otherwise use Device Core's temporary-file execution path. The fixture namespace and LCD resources are released after success, failure or cancellation. Memory, camera and display checks therefore do not require a writable filesystem.

Fixture source transfer uses UTF-8 byte offsets and acknowledgements containing the session nonce, offset and committed content. Repeating an identical chunk is safe; a conflicting chunk or invalid acknowledgement fails immediately. The loader verifies the assembled byte count and SHA-256 before installation, then requires an acknowledgement of completed installation, callable entry points and released source buffers. Only `TIMEOUT` permits recovery, with at most three attempts per bootstrap operation through the public Python API and unchanged per-call/overall deadlines. A lost response after completed installation can recover without re-executing the fixture; an interrupted, partially completed installation fails and is never replayed. Cancellation and semantic errors are not retried.

Bootstrap retries and failures report the phase (`init`, `chunk`, `assemble`, `verify` or `install`), zero-based command index, attempt, encoded input bytes, elapsed milliseconds and last acknowledged operation; chunk diagnostics also include zero-based chunk index, UTF-8 offset and byte count. Capability inspection and host source-read failures are labelled separately. A public execution timeout can include SDK synchronization as well as command execution; these diagnostics locate the bootstrap operation but do not establish a transport or firmware root cause. Reports contain no raw Python output or source text.

An unavailable fixture produces one `fixture-bootstrap` failure. Every dependent check, including the first affected check, reports `BLOCKED` with `blockedBy: fixture-bootstrap` and `FIXTURE_UNAVAILABLE`; independent identity checks still run. The summary counts `passed`, `failed`, `skipped` and `blocked` separately. Blocked checks are neither executed failures nor unsupported-capability skips, and the hardware job fails even without `--require-all`. Cleanup uses an independent cancellation signal; a cleanup failure is reported separately and does not replace the original bootstrap failure or cancellation.

The `espdl_inference` check transfers the repository's 495,840-byte ESPDet Pico Face model through bounded public `python.execute()` calls into a read-only RAM VFS, verifies SHA-256 on both host and device, and invokes the real `espdl.ESPDet` on RGB565 camera frames. It checks repeatability on the same copied frame, detection tuple shape, box overlap, confidence/class ranges and positive inference timing. Reports include the model SHA-256, transfer chunk count/time, load time, frame/detection counts, and mean/worst inference latency. An empty detection list is valid when no face is visible; this sanity check does not establish model accuracy or a latency target. The model, RAM mount and camera are released after success, failure or cancellation; no Flash/SD files or MSC writer leases are used.

Model uploads acknowledge a namespace nonce and exact committed byte offsets. Only a timed-out block may be retried, with at most three public execution attempts; replay accepts only the latest identical block. Conflicting/out-of-order data, missing or incorrect acknowledgements, and cancellation fail immediately. Reports include `modelRetries` and `modelRetryOffsets`. Mounting, model cleanup and final namespace absence require explicit acknowledgements/readback, including when Python prints an exception but the execution wrapper returns success. Existing per-command and overall deadlines remain unchanged.

The default model SHA-256 is `04eec2220adbf642a843c953817fb219a06c094aedb76c181ba2a958c65201dc`. `--model MODEL.espdl` selects another compatible ESPDet model (at most 2 MiB); add `--model-sha256 HEX` to pin its expected bytes. Installed npm packages do not bundle model assets: supply `--model` there. A missing model skips optional acceptance but fails `--require-all`. Camera/ESP-DL support is required for strict acceptance, independently of persistent storage availability.

Storage is separate evidence. The identity check reports volume presence/mount/writer state; a runtime pass does not prove file upload works. Use `--write` on a provisioned board to require mounted writable Flash, perform a unique binary file write/readback and remove only the test's own files. Unmounted or unavailable storage fails that explicit check with `STORAGE_UNAVAILABLE`. Neither CI nor the hardware CLI formats storage, erases data to hide a mount failure, or substitutes an SD volume for Flash.

Camera closed-loop evidence requires a new complete stdout marker or validated JPEG in the current session. Core handles upload, execution and correlation. Checks still cover pixel formats, frame sizes, orientation, IPA state, snapshot rate, preview, image operations and display. Handshake, upload ACK, queued execution or a REPL prompt alone do not prove application completion. Keep receiving during long operations and handle `VM_BUSY` / `VM_TIMEOUT`.

LCD screenshots and test patterns

`--screen-output` displays a known test pattern, calls `device.capture({ source: 'screen', format: 'jpeg' })` to retrieve a JPEG of the last LCD RGB565 frame, and saves it to the requested path. Set `ESP_VISION_WORK_DIR` to a writable workspace outside the checkout, then create the output directory:

```
mkdir -p "$ESP_VISION_WORK_DIR/hardware-validation"
npm --prefix host/device-core run test:hardware -- --serial <USJ-port> --role usj -
↪-checks display --screen-output "$ESP_VISION_WORK_DIR/hardware-validation/lcd.jpg
↪"
```

Capture invokes firmware `debug.capture_screen` in the VM task and reads the display buffer without camera calls. Verify JPEG decoding, dimensions and expected pattern, then compare with the actual device screen. An uninitialized display returns `DISPLAY_NOT_READY`; absent LCD or unsupported firmware returns `NOT_SUPPORTED`.

Never substitute camera capture to hide failure. The screenshot represents the last software-submitted display frame; it does not prove physical panel, cable or backlight operation.

Checks stop current Python programs and may change display content. Schedule exclusive device use rather than sharing ownership with an IDE or another test. Release a valid owner or wait for its lease; switching interfaces does not bypass it. Runtime tests clean their RAM namespace; explicit file tests use unique temporary files and clean their own files. Save reports, JPEGs and logs with actual installed firmware, board and interface. Preserve unknown business completion after reset, disconnection or cancellation instead of replaying writes.

Test entry points and responsibilities

Table 19: One protocol implementation, layered validation

Entry point	Coverage
`host/device-core: npm test`	Deterministic host tests of public API, protocol, sessions, discovery and flashing contracts.
`host/device-core: npm run test:hardware`	Real serial/TCP, camera and display closed loops through the same public Core.
`tools/ci/flash_and_test.py`	CI board/IDF artifact selection and flashing followed by Core hardware checks; no separate EV-MUX implementation.
`tools/tests/`	Firmware native-host state-machine, JSON, queue, storage and reset contracts.

Troubleshooting sequence

Table 20: Symptoms and next checks

Symptom	Check first
Serial text but hello timeout	Raw startup/panic output, wrong USB interface, startup partial frame, runtime mux disablement, CRC/byte lengths and continuous reader.
Configured USB cannot acquire	Capabilities and valid owner; DTR=false does not mean v4 unavailable.
`SESSION_BUSY`	Which IDE/Agent/terminal is renewing; release or wait for lease rather than changing routes to force acquisition.
vmReady=true but file/capture timeout	Running boot/main.py, busy VM queue and actual new REPL marker after interrupt.
Same-seq data but RPC incomplete	Matching type/channel/method/session/epoch; event must not substitute for rsp.
Preview occasionally skips frames	Complete current-scope JPEG, slow readers and expected latest-frame behavior.
Both TCP slots exhausted after disconnect	Separate handshake/lease/30-second idle states and reclaimed old generations, not just released owner.
Ports disappear after download request	ROM re-enumeration, ROM output on USJ, masked download-bit setting/clearing and invalidated handles.
Global drops grow but TCP marker passes	Unread default/fallback interfaces, queue classes and sample scope; other-endpoint counters do not equal this business failure.

Store only necessary state/scalar counters. SSID/password, file content and Flash/NVS backups can contain sensitive information and should not enter public reports. Placeholder examples are not actual configuration.

Keeping documentation aligned with implementation

New channels/backends/actions or stderr/ownership/limit changes require updates to related pages, capabilities, Core and contract tests. Renaming diagram labels alone is not a protocol change; methods need real firmware dispatch implementations.

Firmware unit tests and Core acceptance

`tools/tests/test_ev_*.py` compiles/runs production C logic on the host with ESP-IDF/FreeRTOS fakes; it does not connect to a device or implement another host protocol client. Keep deterministic coverage of partial writes, queue pressure, allocation failures, stale connection generations, session expiry and storage handoff. The accompanying C harnesses and stub headers are test infrastructure, not alternative device-operation tools.

Device Core owns application-visible and hardware acceptance tests. Use its public API for new connection, Python, file, preview and capture scenarios. Firmware fault injection stays alongside firmware because a normal SDK request cannot reliably force task scheduling, memory allocation or driver failures. Source-text assertions are supplementary checks, not substitutes for executing behavior or real hardware acceptance.

11.14 release target acceptance

The release target jobs for Function EV, P4 Vision and S31 Korvo now require the following checks in addition to their board-specific camera, image, display and inference cases. Unsupported selected checks fail with `--require-all`; a missing board or peripheral is not a passing result.

- `--verify-flash` independently compares every image in `flasher_args.json` against Flash using esptool, including the same image-header settings used when writing. A failed comparison prevents runtime tests. Multiple serial candidates require an explicit `ESPPORT`/`--port` before any write; the selected port is retained for verification and runtime acquisition.
- `discovery` enumerates the selected serial endpoint through passive SDK inventory while a nonce-scoped Python variable is live, then verifies the variable and runtime identity remain intact. On an explicitly selected TCP endpoint it instead requires a matching mDNS advertisement and device ID. Enumeration never probes other devices. The standard serial CI jobs do not establish LAN discovery coverage.
- `communication` checks five distinct Unicode output streams, each larger than 4 KiB in UTF-8, with an identity/runtime RPC between executions. Truncated, stale or altered output fails.
- `reconnect` releases and reacquires the selected transport three times, checking device ID, board, firmware version and a new Python marker each time. This tests orderly disconnect/reconnect, not cable removal, power loss or lease contention.
- `modules` executes known-answer SHA-256, hex/base64 binary roundtrips, struct byte ordering, Unicode JSON, BytesIO seek/read and garbage collection. `ulab` computes and checks mean, elementwise multiplication/sum, square root and sorting on the target. Both use RAM without requiring Flash/SD writes.

`--junit PATH` on the Python driver produces GitLab test cases for the individual SDK checks, flash/verify stages and runtime readiness. Bootstrap progress events remain diagnostic log entries; blocked checks appear as failures with their prerequisite recorded. Failures before runtime acceptance produce a failing driver case. Logs and camera/screen artifacts remain available; a successful flash stage alone does not qualify a release.

For the 2026.11.14 candidate, retain the final commit/pipeline, board, interface, firmware version and image artifacts together. Run the full CI firmware matrix only after implementation is complete. The existing publication gate still requires all three target jobs. Tests on an older installed firmware validate the harness only; they do not qualify the candidate. Tag creation and publication are separate release actions.

For coordinated npm releases, CLI packaging on protected master requires successful Core publication or confirmed availability of that version. It resolves the published archive and SHA-512 integrity before `npm ci`, builds and tests the CLI, then publishes that exact checked tarball with the resolved lock. MR checks may use the matching source Core tarball while publication is pending. Plugin release packaging requires the published CLI to bundle the Core version recorded in the Plugin manifest; it does not fall back to an older CLI or Core. Core 0.8.1 must therefore be available before CLI 0.2.1, and both must be available before packaging Plugin 0.2.0. Firmware qualification, npm publication and online Knowledge availability are separate acceptance milestones; a firmware tag alone does not prove that its Knowledge is online.

Core 0.8.1, CLI 0.2.1, Plugin 0.2.0 and Knowledge MCP 0.2.0 are already published to npm. Knowledge MCP 0.2.0 is independent of Core/CLI pins. Its initial retained catalog covers 2026.10.21, 2026.10.23 and 2026.11.06; it does not yet cover the 2026.11.14 firmware candidate. Publish the new firmware snapshot after the tag qualifies and its binaries ship, then import it through a separate reviewed Knowledge release. The Plugin 0.2.0 publication gate waits for any concurrent Knowledge deployment and verifies the live versioned endpoint before writing to npm.

Table 21: Find the relevant topic

Question	Entry point
How do Python print, C printf and IDF logs differ?	*Architecture, Execution Contexts and Standard I/O*
What are the two USB interfaces, UART and Wi-Fi/TCP?	*Physical interfaces, connection generations and build configuration*
How do competing hosts, disconnects and resets recover?	*Session ownership, acquisition and recovery*
How do I implement framing and RPC?	*Frame format, logical channels and RPC protocol*
How do I input Python, transfer files and run long scripts?	*Interactive Python, scripts and file operations*
How do preview and debug.capture_frame differ?	*Preview, debug capture, backpressure and statistics*
How do I provision Wi-Fi and handle startup data?	*Host integration, Wi-Fi configuration and recovery*
How do I install Device Core and integrate serial/TCP, files and preview?	*Device Core integration guide*
How do I prove completion and diagnose failures?	*Communication acceptance, closed-loop tests and troubleshooting*

9.3 Six distinct concepts

- **Physical interface / sink**: backend type, for example `usj` or `tcp`.
- **link**: one exact connection with backend, slot and generation; reusing a TCP slot still creates a new link.
- **stream**: internal `user`/`debug` routing groups, not eight independent serial ports.
- **channel**: frame semantics, for example `repl.stdout` or `preview.frame`.
- **sessionId**: ownership credential tied to an exact link; it cannot move to another socket.
- **vmEpoch**: MicroPython generation. A surviving session does not preserve old-VM work.

9.4 Current implementation boundaries

There is at most one session owner holding both streams. Other links can discover the device, but cannot run a second independent control session. Acquisition contention and explicit handoff are supported; forced eviction of a valid owner is not.

TCP is a raw STA IPv4 byte server on default port 3232 with at most two accepted links. It is not HTTP, WebSocket or MicroPython WebREPL and does not add authentication/encryption in this version. Browsers use a host-side byte bridge: see *Host integration, Wi-Fi configuration and recovery*.

ROM download and EV-MUX runtime are separate protocol phases. The MicroPython REPL remains the interpreter console; IDF system console carries early initialization, ROM/bootloader and some fatal output. Raw text at these boundaries is not a valid business frame.

Chapter 10

Device Core host SDK

`@esp-vision/device-core` is the UI-independent JavaScript host SDK for ESP-VISION EV-MUX v4 firmware. IDE, the Device CLI and custom host applications share its byte framing, ownership, keepalive, Python, files and image handling. Agent workflows use the CLI for device control; the official Knowledge MCP provides documentation over HTTP. SDK source is maintained in `host/device-core`.

10.1 Application API reference

This page covers the complete object returned by `createDevice` in package 0.4.0 (prepared, not published by this change). Unless stated otherwise, operations are asynchronous and reject on failure. Unknown request fields are rejected. Do not add an undocumented option such as `timeout` to every method: accepted keys are listed per method. Millisecond timeouts are positive finite numbers.

Method	Purpose
`createDevice(options)`	Create one independently managed device object.
`device.discover(options)`	List LAN advertisements without acquiring ownership.
`device.connect(options)`	Open/negotiate a v4 connection and probe REPL readiness.
`device.disconnect()`	Release ownership, cancel work and close transport.
`device.inspect(request)`	Read identity, capabilities, runtime, network or storage.
`device.configure(request)`	Apply a network configuration patch.
`device.control(request)`	Stop, reset or enter ROM download mode.
`device.firmware.check(request)`	Check the recommended firmware for the board.
`device.firmware.update(request)`	Download, qualify, flash and verify application or complete-install images.
`device.capture(request)`	Capture camera or LCD JPEG.
`device.on(event, handler)`	Subscribe and obtain an unsubscribe function.
`device.python.input(data)`	Send interactive REPL/stdin input.
`device.python.execute(request)`	Execute source or an existing device script.
`device.files.list(request)`	List a directory.
`device.files.read(request)`	Read bytes or UTF-8 text.
`device.files.write(request)`	Create/overwrite a text or binary file.
`device.files.remove(request)`	Remove one file or empty directory.
`device.files.mkdir(request)`	Create a directory and optional parents.

10.1.1 Creation and byte transport

`createDevice({ transport?, discovery? } = {}) → device`

`transport` supplies raw bytes; `discovery` supplies `list({ timeout, signal }) → Promise<Array>`. Either can be omitted at construction; a transport is required at connection time. A discovery-only instance may receive a transport in `connect`. Core does not choose a serial port or create a socket from a discovery record automatically.

Transport member	Contract
`writeBytes(bytes)`	Required; accepts Uint8Array, returns a Promise after all bytes are accepted by the host connection. Reject short/failed writes.
`subscribe({ data, error, close })`	Required; synchronously returns an unsubscribe function. data receives original Uint8Array chunks.
`connect({ signal })`	Optional; opens the connection after Core subscribes.
`disconnect()`	Optional; closes the physical connection and releases reader/writer resources.

10.1.2 device.discover(options)

```
{ timeout = 3000, signal? } → Promise<DiscoveredDevice[]>
```

`timeout` is an integer from 50 to 10000. A discovery provider is mandatory; missing configuration throws `DISCOVERY_UNAVAILABLE`. Results are normalized and sorted by hostname; at most 256 records are accepted. Discovery neither connects nor determines whether an owner is active.

```
const discovered = {
  deviceId: 'e8f60ae75642',
  hostname: 'esp-vision-esp32-p4x-eye-e75642-2026-08-27',
  board: 'ESP32_P4X_EYE', firmwareVersion: '2026.08.27',
  evMuxVersion: 4, compatible: true, port: 3232,
  addresses: ['192.168.1.107'],
  // Optional: connectionUrl (bridge), expiresAt (expiry timestamp).
}
```

The example is illustrative metadata, not a device to connect to. `deviceId` is the main-chip MAC without separators. `firmwareVersion` is the firmware version, not the IDE version. Build a byte transport using the selected address/port, then pass the record as `expectedDevice`. Node and browser providers are documented in *Advanced export API reference*.

10.1.3 device.connect(options)

```
{ transport?, role = "unknown", expectedDevice = null, timeout = 60000,
probeRepl = true, autoBaud = true, signal? } → Promise<Inspection>
```

`role` accepts `unknown`, `usj`, `usb`, `uart` or `tcp`. Core subscribes before opening transport, negotiates v4, checks `expectedDevice` when provided, acquires a session and probes the REPL. The result is the default inspection (identity/capabilities/runtime). An already connecting, connected or disconnecting instance rejects with `DEVICE_BUSY`. Identity matching is not authentication. The default `probeRepl: true` sends an interrupt and probes the REPL, which can stop an existing Python program. Set `probeRepl: false` to acquire the EV-MUX session without sending REPL input, for observation or native RPC operations while a program continues running. This still acquires the device session and respects ownership; it is not a force-takeover option. Python execution and REPL-backed file operations can subsequently interrupt the program. The option must be a boolean.

`autoBaud` defaults to `true`. When the firmware advertises UART negotiation and the actual session owner is `uart`, Core uses the transport's optional `setBaudRate(baudRate, { signal })` to test 2000000, 1500000, 921600, 460800 and 115200 bps, stopping at the first verified result. `autoBaud: false` keeps short identity-only probes at 115200 without running speed tests. Native USB and legacy firmware skip this negotiation. The option must be boolean. Runtime and ROM/stub flashing speeds are independent.

10.1.4 device.disconnect()

Returns a Promise. It releases the session, aborts pending work, closes the session interpreter, unsubscribes input and calls the optional transport close. Repeated calls share cleanup; an already disconnected object resolves without work. Release/transport cleanup failures may reject, so use `finally` even when connect fails. Reconnection requires a transport that can reopen or a new transport supplied to `connect`.

10.1.5 device.inspect(request)

```
{ fields = ["identity", "capabilities", "runtime"], timeout = 5000, signal?
} → Promise<Inspection>
```

Field	Result
`identity`	`{ deviceId, hostname, board, firmwareVersion } | null`
`capabilities`	Negotiated firmware capabilities, features, channels and limits.
`runtime`	Session/parser/transport snapshot plus vmReady, vmEpoch and current run or null. `scriptRunning` and `run` track executions started by this SDK session; a fresh connection cannot use false/null to establish that a detached program is idle. `vmReady` means the VM has started, not that it is idle.
`network`	network.status reply including enabled/configured/state/IPv4 and operation state; no saved password.
`storage`	flash and sd storage.writer.status replies; requires storage.writer.lease. New firmware includes fs_state/fs_error/fatfs_result/format_supported; see *Storage, Filesystems and Data Protection*. Writer ownership alone does not establish filesystem readiness or authorize formatting.

`fields` must be a nonempty array of the five names above. Only requested keys are returned; duplicate fields are queried once. Identity/capabilities are copied negotiated data, not a new sensor sample. Network and storage perform RPCs; the supplied timeout applies to each underlying query, not a single aggregate inspection deadline.

10.1.6 device.configure(request)

`{ network: { enabled?, credentials? }, timeout = 45000, signal? } → Promise<{ ok, completed, network }>`

`credentials` is `{ ssid, password }` or `null`. SSID is 1–32 UTF-8 bytes without NUL. Password is empty for an open network, 8–63 printable ASCII characters, or a 64-digit hexadecimal PSK. `enabled` is boolean. Supply at least one patch field; omitted fields preserve existing settings. Clearing credentials disables the network and cannot be combined with `enabled: true`.

```
await device.configure({ network: {
  credentials: { ssid: 'example', password: 'example-password' },
  enabled: true,
} })
await device.configure({ network: { enabled: false } })
await device.configure({ network: { credentials: null } })
```

The total configuration deadline includes polling operation completion, not only the accepted ACK. `completed` contains acknowledged `{ action, operationId }` steps. Failure attaches `error.completed` and optionally `error.pending`; already completed steps are not rolled back. The patch is a sequence, not an atomic transaction. Prefer USB provisioning when Wi-Fi changes can drop TCP. Never log credentials.

10.1.7 device.control(request)

`{ action, mode?, timeout = 15000, signal? } → Promise<control result>`

Request	Behavior
`{ action: "stop" }`	Interrupt/stop Python; usable while python.execute is active.
`{ action: "reset", mode: "soft" }`	Soft-reset the VM; observe the new epoch and readiness.
`{ action: "reset", mode: "hard" }`	Restart hardware; the connection may disappear.
`{ action: "bootloader" }`	Request ROM download mode; disconnect and use a ROM loader.

`mode` is mandatory only for reset and rejected for other actions. A stop acknowledgement does not prove the running script has finished cleanup: observe its `done` result. Reset/download ACKs do not prove normal boot or download enumeration; reconnect and inspect after the device returns.

10.1.8 device.python.input(data)

Accepts a string or Uint8Array, returns the underlying write Promise. Send `\r` to submit an interactive line. During a `completion: "start"` run, it writes stdin (for example Python `input()`). It has no request-object timeout/signal arguments. Output is delivered by `python.output`; writing input alone does not prove evaluation succeeded. Other automatic operations reserve the interpreter and reject conflicting terminal input with `DEVICE_BUSY`.

10.1.9 device.python.execute(request)

`{ source? | path?, completion = "wait", timeout = null, maxOutputChars?, signal? }`

Provide exactly one of `source` (string) or `path` (absolute device file). With `completion: "wait"`, resolves `{ ok, path, output, outputTruncated }`; inline source has `path: null`. With `completion: "start"`, resolves `{ ok: true, path, done }` and `done` resolves the same result shape. The start result follows a private VM-entry marker from the generated runner; it does not confirm application initialization or completion. `timeout` is accepted only for wait mode; omission or null means no aggregate execution deadline, while setup exchanges remain bounded. `maxOutputChars` is a nonnegative integer controlling retained output, not UART bandwidth or script lifetime.

```
const off = device.on('python.output', text => terminal.write(text))
try {
  const run = await device.python.execute({
    source: 'name = input("Name? "); print(name)', completion: 'start',
  })
  await device.python.input('example\r')
  console.log(await run.done)
} finally { off() }
```

Here `terminal` is supplied by the UI. Start mode uploads source to a unique Core-owned temporary file and removes it after completion; it does not remove a user-supplied path. After the start handle returns, `device.disconnect()` releases the session without requesting a stop; `done` rejects locally as the session closes, and the device program may continue. The request's `signal` remains bound until `done` settles and requests a stop if aborted while the session is available. Disconnect can prevent temporary-file cleanup. Python stdout remains tied to the original EV-MUX session and cannot be reattached from a new connection; keep the session open for live output and use captures or application progress artifacts to check a detached program. Inspect the returned output for Python tracebacks: `ok` is an execution/transport result and is not a universal assertion that arbitrary Python code produced a correct application result.

10.1.10 Device file operations

Device paths must be absolute and at most 123 UTF-8 bytes. Control characters, backslash, double quote, `..` anywhere, repeated slash and a trailing slash are rejected. Only list accepts root `/`. Filesystem mount/storage ownership and free space must permit the operation. Methods below reserve the same interpreter used by Python.

device.files.list(request = {})

`{ path = "/", recursive = false, signal? }`

Returns a flat array of { name, path, type: "file" | "directory", size }; directory size is zero. Recursive results remain flat.

device.files.read(request)

```
{ path, encoding = "binary", timeout = null, chunkTimeout = 30000, maxBytes
= null, signal? }
```

Returns Uint8Array for binary or string for utf-8. Omitted/null `timeout` means no aggregate read deadline; `chunkTimeout` bounds each REPL exchange and defaults to 30000 ms. An explicit positive total timeout caps each exchange by the remaining time. maxBytes is null or an integer 0..16777216 and bounds the complete file read; exceeding it rejects instead of returning a silent prefix.

device.files.write(request)

```
{ path, data, overwrite = true, onProgress = null, signal? }
```

data is string or Uint8Array. overwrite:false requests create-only. Progress receives { writtenBytes, totalBytes }. Returns the verified commit acknowledgement with total bytes, complete:true and path, after the firmware checks the complete SHA-256. It does not execute the file.

device.files.remove(request)

```
{ path, type = "file", signal? }
```

type is file or directory; directory removal requires an empty directory. Returns Promise<void>. There is no recursive remove flag.

device.files.mkdir(request)

```
{ path, recursive = false, signal? }
```

Returns {ok:true}; recursive mode creates parents and accepts an existing directory, but not an existing file in its place.

SDK 0.2.0 and later transport all file types through reliable raw binary `file.write` frames, including Python source and model files. Upload requires firmware feature `file.write.binary` and a valid `limits.fileWriteChunkMax` (4096..16384, aligned to 4096); missing support rejects with UNSUPPORTED_OPERATION and a firmware-upgrade message. There is no Base64/JSON or REPL upload fallback. list/read/remove/mkdir still use framed REPL; no separate public files.writeBinary method is needed. Progress reports confirmed bytes, not a durable commit. Only the final resolved result means length/hash verification and file publication completed. Cancellation or failure triggers best-effort abort; firmware also expires stale transfers and cleans up on session/VM changes. Existing target contents survive upload, hash and ordinary pre-publication failures. FAT replacement uses an old-file backup and rollback for ordinary rename failure; it is not a power-loss-atomic journal. See *Interactive Python, scripts and file operations* for the wire contract. Application updates use `device.firmware.update()` and reuse the ROM implementation in the `/firmware` subpath.

10.1.11 device.capture(request)

```
{ source, format = "jpeg", quality?, timeout = 15000, signal? } → Promise<{
data, mimeType, metadata }>
```

`source` is `camera` or `screen`; the only format is `jpeg`. Optional quality is an integer 1–100; omission uses the firmware default. data is Uint8Array, mimeType is image/jpeg, and dimensions are in metadata.width/height (not top-level width/height).

```
const frame = await device.capture({ source: 'screen', quality: 80 })
console.log(frame.metadata.width, frame.metadata.height, frame.data.length)
const blob = new Blob([frame.data], { type: frame.mimeType })
```

Camera capture requires both `debug.capture_frame` and `debug.capture_frame.observed.v1`. It encodes the independent frame published by the last successful application snapshot in the VM; it never initializes the camera, acquires a new physical frame or overwrites the application's image. The observation contains the original captured pixels, not later application edits. Core rejects unsafe older firmware locally with `UNSUPPORTED_OPERATION` and an upgrade message, including direct session RPC/action capture paths. No frame or an invalidated observation after camera reconfiguration, shutdown or VM reset returns `CAMERA_FRAME_NOT_READY`. Observation-copy allocation failure returns `NO_MEM` while preserving the application snapshot success. Timeout and cancellation do not trigger a capture fallback. Use `connect({ probeRepl: false })` when observing a running program.

Screen capture requires an initialized LCD and reads its last software-submitted RGB565 frame without starting a camera or redrawing the screen. Unsupported hardware reports NOT_SUPPORTED; uninitialized LCD reports DISPLAY_NOT_READY. Screenshots do not read back the physical panel. See *preview-debug* for lazy frame storage, DMA completion and preview delivery.

10.1.12 device.on(event, handler)

Synchronous; returns `unsubscribe()`. Subscribe before connect when startup output is needed. Throwing listeners do not interrupt parsing or other listeners; a diagnostic listener.error is emitted. Remove subscriptions and revoke image object URLs when disposing UI.

Event	Payload	Meaning
`python.output`	`string`	Python stdout/stderr, prompts and interactive output.
`raw`	`string`	Text outside EV-MUX frames, including early boot output.
`log`	`string`	Firmware/IDF diagnostic text.
`preview`	`{ data, mimeType, metadata }`	JPEG from application img.flush(); delivery may drop frames.
`state`	`{ state, identity?, reason? }`	connecting/ready/disconnected and hello/network.changed hints; identity is provided on ready.
`diagnostic`	`{ event, ...details }`	rpc.request/response/timeout, listener.error and cleanup diagnostics; sensitive network credentials are redacted.
`rpc`	`{ channel, metadata, payload }`	Advanced inbound RPC observation; prefer typed device methods for operations.

10.1.13 Concurrency, cancellation and errors

Inspection, configuration, control, Python, file and capture methods require a ready connection. Discovery, event subscription and firmware checks do not; firmware update can also run disconnected with an explicit board and ROM loader. The facade serializes exclusive Python/file/configuration/capture operations by rejecting conflicts with DEVICE_BUSY; it does not silently enqueue arbitrary application work. stop/reset can coordinate with an active python.execute. A started run keeps its reservation until done; interactive input is allowed while that run accepts stdin.

Error	Handling
`TypeError / RangeError / INVALID_PATH`	Correct parameters; requests reject unknown keys, unsupported enums and invalid paths.
`DEVICE_NOT_READY / DEVICE_BUSY`	Connect first, or await/stop the current operation.
`DISCOVERY_UNAVAILABLE / UNSUPPORTED_OPERATION`	Supply a provider or use compatible firmware capabilities.
`SESSION_BUSY / SESSION_LOST / VERSION_MISMATCH`	Release another valid owner, reconnect after loss, or update firmware; do not force takeover or retry writes blindly.
`TIMEOUT / AbortError / CONNECTION_CLOSED`	AbortSignal may preserve its supplied Error reason. Completion can be unknown; verify identity/state/files before retrying changes.
`VM_BUSY / VM_TIMEOUT / DISPLAY_NOT_READY / NOT_SUPPORTED`	Handle firmware errors explicitly. Stop/initialize the required resource or report unavailable hardware.

All asynchronous callers must catch rejections; errors may include code, errno, completed and pending. Not every error class supplies a code. Disconnect/cancel does not roll back device changes, guarantee script termination, or authorize retrying a non-idempotent operation.

10.1.14 Firmware checks and updates

`device.firmware.check({ catalog?, identity?, version?, mode = 'update', fetch = globalThis.fetch, signal? })`

Returns `{ available, currentVersion, release, releases }`. Uses the connected device identity; when disconnected, pass an inspected `identity` containing `board` and `firmwareVersion`. Checking performs no connection acquisition, reset or writes. `catalog` is either an in-memory object or an absolute HTTPS JSON URL. Fetch errors reject; no matching board returns `release: null, available: false`. Without catalog, Core reads the official dl.espressif.com ESP-VISION index and per-version update metadata; a custom catalog remains an advanced override. Unqualified historical entries remain visible but cannot be selected for installation.

Catalog shape: `{ schemaVersion: 1, releases: [{ board, version, image }] }`. The publisher orders entries newest first; By default Core selects the first entry matching the board and validates its v4 image metadata; an explicit version overrides this selection. `available` means its version differs from the current firmware, not that an arbitrary calendar/git label compares numerically greater. The feed publisher is responsible for the recommended release order; the UI must show the target version before an update, including any intended downgrade.

Each `image` contains `evMuxVersion: 4`, non-negative integer `chipFamily`, `offset` plus either exact `chipRevision` or integer `chipRevisionMin`/`chipRevisionMax` bounds, positive integer `bytes` and `partitionSize` (bytes must fit), lowercase 64-character `sha256`, and `partitionTable: { offset, bytes, sha256 }`. When downloading rather than supplying bytes, add an absolute HTTPS `url` pointing directly to the application bin. A zip/artifact download is not a raw bin and is not extracted by Core. Board names must exactly match the firmware identity. Version labels refer to firmware, not SDK/IDE versions. SHA-256 detects mismatch against the selected feed; it does not authenticate a malicious feed.

`device.firmware.update({ release, board?, mode = 'update', data?, fetch = globalThis.fetch, openLoader, reset?, onProgress?, onPhase?, signal? })`

`release` is a selected check result entry. A connected device must match its board; without a connection, explicitly pass the physically selected `board`. Without `image.source`, `data` is a Uint8Array application BIN; otherwise Core downloads `release.image.url`. With `image.source`, both downloaded and supplied `data` are the complete BIN container; Core validates and extracts the application internally as described below. Core validates size/hash before releasing the EV-MUX connection, then calls host `openLoader({ signal })` to acquire the ROM download port. The host adapter handles serial/WebUSB selection and ROM entry; Core does not implement

network OTA or invent a second download-mode wire command. Use a local download connection even when discovery/inspection used TCP.

The loader contract is the existing *Advanced export API reference* FirmwareFlashSession contract. Optional `reset(loader)` replaces its hard-reset policy; `onProgress(percent)` reports 0–100 including write verification and `onPhase(text)` reports phases. In default update mode only the application partition is written: chip/revision, security mode and the existing partition table are checked before writing, then the built-in loader verifies the whole write using device-side MD5. Custom loaders without a verified-write receipt retain complete SHA-256 readback. Partition mismatch rejects; default update mode does not replace the partition layout or erase NVS/user files. Explicit installation is described below.

Returns `{ verified: true, needsReset, version, board, reconnected: false }` only after firmware write verification. `needsReset: true` means the host must ask for a physical reset. No automatic EV-MUX reconnection or running-version claim is made: reconnect and inspect the firmware version separately. Cancellation applies before download/write, including loader acquisition; after the flash write starts it is deliberately deferred so verification can finish. A failed update rejects and closes the acquired loader; rollback is not promised. Connection attempts, other device operations and concurrent updates are rejected with DEVICE_BUSY during an update. Unknown request fields reject. The device remains disconnected after handoff, including failures after that point.

```
const result = await device.firmware.check({
  catalog: trustedCatalog, // Ordered release metadata from the host publisher.
  // For offline/ROM recovery only: identity: previouslyInspectedIdentity
})
if (result.available && userApprovedVersion === result.release.version) {
  const updated = await device.firmware.update({
    release: result.release,
    openLoader: ({ signal }) => host.openRomLoader({ signal }),
    onProgress: percent => console.log(percent),
  })
  console.log(updated) // Reconnect/inspect separately; prompt if needsReset.
}
```

Complete installation on blank devices

Use the same two firmware methods with `mode: 'install'`. No EV-MUX connection or running firmware is required. Choose the physical board explicitly, enter ROM download mode and select its ROM-capable USB-Serial-JTAG (USJ) or UART port. An application-provided USB-OTG CDC interface cannot be assumed on a blank board. The host still supplies the ROM adapter; CLI includes its Node/esptool adapter. This is local ROM flashing, not TCP OTA.

Default `mode: 'update'` retains the application-only contract. Core never changes to install mode automatically after a layout mismatch. Starting with Core 0.7.0, explicit install deletes internal Flash filesystem files, including scripts and models, then writes a qualified merged image at address zero, including bootloader, partition table and application. NVS configuration, including saved Wi-Fi settings, may also be replaced by that image. SD cards are not erased. Hosts must clearly disclose this deletion before users select complete installation. There is no whole-chip erase, firmware backup, automatic restore or rollback.

The existing schemaVersion 1 release entry gains optional `install: { offset: 0, bytes, flashSize, sha256, url, chipRevisionMin, chipRevisionMax, bootloader: { offset, bytes, sha256 } }`. `flashSize` is the layout's required flash capacity in bytes. Bootloader revision bounds are intersected with the application's bounds by the release publisher. `image` remains the application descriptor; install validation checks that its application and partition-table hashes and the bootloader hash match their exact ranges inside the merged image. All ranges must fit, be ordered and not overlap. Future release pipelines generate this metadata from the existing full image and real IDF artifacts; historical images without approved installation metadata are never guessed compatible.

`firmware.check({ mode: 'install', ... })` selects the first qualified installation release, or an exact requested version. App-only releases remain visible in `releases`; selecting one explicitly rejects with FIRMWARE_INSTALL_UNAVAILABLE. Unknown modes reject. `firmware.update({ mode: 'install', ... })` validates all image bytes and parses the authenticated ESP partition table before releasing a

session or opening a ROM port. It checks partition bounds, overlap, labels and any MD5 record, requires a matching application partition and a sector-aligned writable unencrypted `ffat` data/FAT partition outside the firmware write range, then checks the actual chip, silicon revision, security state and JEDEC capacity. Missing/unsupported layouts reject; legacy `vfs` is not guessed to be FAT. After qualification it erases only that `ffat` range. A successful erase command leads directly to firmware writing, without erased-range reads, sampling or digest calculation. Erase failures and cancellation before writing cannot report success or reset the device; uncertain operations are not replayed. Once firmware writing begins, it finishes full image verification and the existing reset policy. Reconnect and inspect to confirm the filesystem mounted and the new firmware actually runs separately.

```
const device = createDevice() // No runtime transport or connect() required.
const selected = await device.firmware.check({
  identity: { board: 'ESP32_P4X_EYE', firmwareVersion: 'unknown' },
  mode: 'install', version: selectedFirmwareVersion,
})
await device.firmware.update({
  release: selected.release, board: 'ESP32_P4X_EYE', mode: 'install',
  openLoader: ({ signal }) => host.openRomLoader({ signal }),
})
```

10.1.15 Selecting a firmware version

`releases` contains all catalog entries for the current board, in publisher order. Without `version`, `release` is the first recommended entry. Pass `version` to select an exact firmware version; a missing version rejects with FIRMWARE_VERSION_UNAVAILABLE, never falls back to another build. Duplicate board/version entries reject as ambiguous. The catalog determines which history is available; Core does not scrape all Git tags, CI artifacts or unpublished versions. Choosing an older version is explicit and uses the same update qualification.

```
const history = await device.firmware.check({ catalog: trustedCatalog })
console.log(history.releases.map(item => item.version))
const selected = await device.firmware.check({
  catalog: trustedCatalog, version: selectedFirmwareVersion,
})
await device.firmware.update({
  release: selected.release,
  openLoader: ({ signal }) => host.openRomLoader({ signal }),
})
```

Node hosts can reuse the shipped ROM adapter directly:

```
import { openNodeRom } from '@esp-vision/device-core/node'
await device.firmware.update({
  release: selected.release, board: 'ESP32_P4X_EYE', mode: 'install',
  openLoader: ({ signal }) => openNodeRom({ port: selectedRomPort, signal }),
})
```

SDK 0.5.1 includes the browser ROM adapter `openBrowserRom({ port, logger?, signal? })` from `@esp-vision/device-core/browser`. It uses the same install/update contract above; see *Integrating the Device Core npm Package* for user-gesture serial permission and bundler requirements.

10.1.16 Immutable release snapshots and single-BIN updates (0.4.0)

SDK 0.5.1 includes these release readers. Import `loadOfficialFirmware` and `loadReleaseManifest` from `@esp-vision/device-core/firmware`. The root still exports only `createDevice`.

`loadOfficialFirmware({ board?, version?, fetch = globalThis.fetch, signal? })` prioritizes `config.toml`'s `[publication]`: `schemaVersion: 1`, `version`, 40-character `sourceCommit`, `manifestUrl`, raw-byte `manifestSha256`, and `releases: [{ version, manifestUrl, manifestSha256, boards }]`. The history has no duplicate versions and each board set is nonempty and unique. Additional publisher transaction fields are preserved but do not change reader behavior. A

selected board/version must exist in that reference. Only the selected manifest is downloaded, never the historical board × version matrix; `firmware.check({ version })` forwards its version to this reader. An explicit invalid publication/reference/manifest rejects, including a 404 or hash mismatch, without fallback. Only configs lacking `publication` use the legacy feed. Explicit versions also restrict legacy metadata requests.

`loadReleaseManifest({ url, sha256, version?, sourceCommit?, boards?, fetch = globalThis.fetch, signal? })` reads one referenced snapshot without consulting mutable config. SHA-256 covers the original UTF-8 JSON bytes. The result has `schemaVersion: 1`, `version`, `sourceCommit`, positive `pipelineId`, `layout: 'legacy' | 'single-bin'`, the existing qualified `releases` array, the complete `website` snapshot and optional `mcp` (null means explicitly absent). Optional identity/board arguments must exactly match; every release and source is validated before returning. Official URLs stay under `https://dl.espressif.com/AE/esp-vision/`; model downloads use `raw.githubusercontent.com/espressif/esp-vision/<sourceCommit>/models/...`. MCP descriptors contain `url/bytes/sha256/version/sourceCommit` and use the official `bin/mcp/` source. Website release/boards/models are checked; hidden firmware boards need not appear in its visual board list. The reader enforces at most 64 release boards, 256 version/model records and 4 MiB of manifest bytes. Hash pinning detects drift from the trusted publication reference; it is not a signature/authentication scheme for an untrusted publisher.

`loadOfficialFirmware` returns the manifest snapshot plus `publication`; optional `board` filters only returned firmware releases after validation of the complete manifest. Its `publication.releases` supplies qualified historical version/board references. `website.releases` may retain older downloadable versions without qualified metadata; never advertise those as flashable merely because a BIN URL can be formed. A website built from a fixed reference should pass that reference directly to `loadReleaseManifest` when selecting a version, keeping UI and firmware choices tied to one snapshot.

In `layout: 'single-bin'` every release has `image.source: { url, bytes, sha256, offset }`. Its URL equals `image.url` and the complete `install.url`; length/hash equal `install.bytes/sha256`. `source.offset` equals `image.offset` and bounds the application inside that complete container. `image.bytes/sha256` continue to describe the application. Core downloads the complete source once, rejects size/hash/embedded bootloader/partition/application mismatch before ROM acquisition, extracts the application, then uses the existing device security/layout/write/verification flow. A changed partition layout still rejects default update; there is no automatic install fallback. Explicit `mode: 'install'` writes/verifies the complete image. Supplied `data` follows the selected descriptor: the full container when `image.source` is present, otherwise the existing raw application BIN. `legacy` manifests and old application-only catalogs remain supported. Downloads are bounded by approved exact size and 256 MiB, use HTTPS, reject redirects and do not use HTTP Range. Browser callers do not download, slice or reimplement verification themselves.

Hash-pinned manifests and complete-BIN sources use browser HTTP cache (`cache: default`); mutable config and legacy app-only BINs retain `no-store`. A fetch-stage TypeError on a pinned asset gets one `no-store` retry for cached responses lacking CORS headers. HTTP, body-read, SHA-256 and metadata failures are not retried or downgraded.

10.2 Advanced export API reference

This reference follows every entry of package.json exports: the public import paths and named exports. It lists every non-underscore public prototype method of the exported classes. Underscore methods and src/device_* implementation modules without a package export are internal. Application developers should start with *Application API reference*, not combine multiple owners/parsers on one connection. Signatures below use JavaScript parameter/default syntax; asynchronous methods return Promises.

10.2.1 @esp-vision/device-core

Only `createDevice(options)` is exported at the root; all parameters and the returned device API are specified in *Application API reference*.

10.2.2 @esp-vision/device-core/evmux

EvMuxParser

```
new EvMuxParser(handlers = {})
```

Incremental byte parser with bounded partial frames, CRC checks, frame filtering and channel dispatch. handlers: text, rawText, log, command, other, preview(base64, metadata, bytes), debugImage(base64, metadata), rpc(metadata, payload), filterFrame(metadata, payload), frameRejected(metadata). A filter returning false rejects a frame before delivery.

EvMuxParser.setLimits({ metadataMax, payloadMax, frameProgressTimeoutMs, frameTotalTimeoutMs } = {})
: Update positive metadata/payload sizes and partial-frame timing limits.

EvMuxParser.resetReplTextDecoders()
: Reset stdout/stderr UTF-8 decoder state.

EvMuxParser.reset()
: Clear buffered data, partial-frame timers and diagnostics.

EvMuxParser.getDiagnostics()
: Return active/rxBytes/rxChunks/bufferedBytes/frames/droppedFrames/invalidImageFrames/lastChannel, incompleteFrameTimeouts, and a detached partialFrame timing snapshot (null or startedAt/lastProgressAt/progressDeadline/totalDeadline in milliseconds). These diagnostics contain no frame payload. reset clears counters and the partial-frame state.

EvMuxParser.hasPartialFrame()
: Return whether buffered bytes form an incomplete possible frame.

EvMuxParser.feedText(text)
: Encode legacy text input and feed the parser; byte input is preferred.

EvMuxParser.feedBytes(bytes)
: Feed arbitrary Uint8Array chunks, including split/coalesced frames.

EvMuxParser.dispatch(metadata, payload)
: Dispatch already parsed metadata/payload to channel handlers; this bypasses wire validation.

makeEvMuxFrame

```
makeEvMuxFrame({ channel, type = 'data', method = null, payload = '', seq = 0, sid = 'web-ide', sessionId = undefined, vmEpoch = undefined, contentType = undefined, width = undefined, height = undefined, route = undefined, encoding = undefined })
```

Return Uint8Array wire bytes including metadata, binary/text payload and CRC. seq is correlation, sessionId/vmEpoch are ownership generations; encoding a frame does not send it or acquire a session.

10.2.3 @esp-vision/device-core/evrepl

EV_REPL_FILE_CHUNK_BYTES

Default chunk size for the legacy REPL file writer, separate from script.write RPC chunks.

```
224
```

EV_REPL_INPUT_MAX_BYTES

Default UTF-8 budget for a complete REPL input command including exec escaping.

```
511
```

EvRepl

```
new EvRepl(io)
```

Framed MicroPython REPL coordinator. io.writeEvmux(channel, type, method, payload) must return a Promise. stdout must be fed continuously. Public helper methods below include interpreter bookkeeping; application callers normally use device.python/files instead.

`EvRepl.execInputBytes(source)`
Return the UTF-8 byte count of the escaped REPL exec command, including command overhead.

`EvRepl.createScriptEndSentinel()`
Create and return a unique script-completion print command.

`EvRepl.createScriptRunCommand(path)`
Create an execfile runner with an inline completion sentinel.

`EvRepl.cancelScriptEndSentinel()`
Discard the outstanding completion marker.

`EvRepl.drainScriptEnd()`
Consume buffered script-completion markers and notify onScriptEnd.

`EvRepl.partialMarkerTail()`
Find a trailing partial completion marker for buffering.

`EvRepl.feedStdout(text)`
Feed text and resolve prompt/output waiters.

`EvRepl.drain()`
Process buffered prompt and completion data.

`EvRepl.failAll(error)`
Reject queued/current waiters with an error.

`EvRepl.resetEpoch(error)`
Reset interpreter epoch state and invalidate old waiters.

`EvRepl.enqueue(operation, { signal = null } = {})`
Serialize operation; returns its result Promise and supports cancellation.

`EvRepl.writeStdin(data)`
Write string/Uint8Array to repl.stdin.

`EvRepl.execLine(line, { timeout = 8000, signal = null } = {})`
Execute a prepared line and await its prompt/output.

`EvRepl.execPy(source, { timeout = 8000, signal = null } = {})`
Escape source into an exec command and return collected output.

`EvRepl.getDeviceInfo({ signal = null } = {})`
Query platform, implementation, board and interpreter information.

`EvRepl.getFsStats(path = '/', { signal = null } = {})`
Query filesystem capacity/free-space information for a path.

`EvRepl.listFs(path, { recursive = false, signal = null } = {})`
Return flat {name,path,type,size} entries; optionally recurse.

`EvRepl.walkFs({ signal = null } = {})`
Return the legacy nested filesystem tree.

`EvRepl.readFile(fn, { timeout = null, chunkTimeout = 30000, signal = null, maxBytes = null } = {})`
Return complete Uint8Array file contents; reject changed, oversized or incomplete reads.

`EvRepl.touchFile(fn)`
Create/truncate one file; Promise<void>.

`EvRepl.removeFile(path, { signal = null } = {})`
Remove one file; Promise<void>.

`EvRepl.removeDir(path, { signal = null } = {})`
Remove an empty directory; Promise<void>.

`EvRepl.makePath(path)`
Create a path through the legacy REPL helper.

`EvRepl.writeFile(fn, data, chunk_size = EV_REPL_FILE_CHUNK_BYTES, direct = false, { signal = null } = {})`
Legacy REPL byte/chunk helper; application files.write and script staging use the binary engine, never this helper.

EvRepl.readSysInfoMD()
Return formatted system information as Markdown.

evReplExecInputBytes

`evReplExecInputBytes(source)`

Return the UTF-8 byte count of the escaped REPL exec command, including command overhead.

10.2.4 @esp-vision/device-core/session

DeviceSession

`new DeviceSession({ transport, EvMuxParser, makeEvMuxFrame, EvRepl, handlers = {}, sid = 'ide-client', sleep = defaultSleep, })`

Low-level v4 interpreter. Inject transport, EvMuxParser, EvRepl and makeEvMuxFrame; handlers receives text/rawText/log/preview/rpc/diagnostic and session lifecycle callbacks. The caller subscribes transport input and feeds bytes itself; constructing this class does not open a connection. Application createDevice does this lifecycle work for you.

DeviceSession.feedBytes(bytes)
Feed received byte chunks into the owned parser.
DeviceSession.feedText(text)
Feed compatibility text input.
DeviceSession.hasFeature(feature)
Return whether negotiated capabilities advertise a feature.
DeviceSession.writeFrame(channel, type, method, payload, { authorize = null, vmRequest = null, seq: requestedSeq = null, signal = null, writeTimeout = null, } = {})
Encode/send a frame with current session/epoch context; returns the transport Promise.
DeviceSession.rpc(method, payload = {}, { timeout = 4000, channel = null, signal = null, binaryResponse = false } = {})
Send a correlated RPC, awaiting JSON or binaryResponse; reject timeout/cancellation/firmware error.
DeviceSession.captureImage(method, payload = {}, options = {})
Call a capture method and return JPEG {data,mimeType,metadata}. Camera methods, including direct `rpc` and request `writeFrame` calls, require `debug.capture_frame.observed.v1` and reject unsafe older firmware before sending bytes. `CAMERA_FRAME_NOT_READY` and `NO_MEM` remain explicit firmware errors; no physical capture fallback is issued.
DeviceSession.negotiate(options = {})
Probe v4, validate identity/limits, acquire ownership and optionally prove REPL readiness.
DeviceSession.writeFile(path, content, { encoding = 'utf-8', mode = 'overwrite', runAfterWrite = false, signal = null, onProgress = null, } = {})
Write UTF-8/binary through the shared reliable file.write binary engine; verify SHA-256 commit before returning. Requires file.write.binary; no upload fallback.
DeviceSession.writeScript(path, content, { runAfterWrite = false, signal = null } = {})
UTF-8 compatibility wrapper over writeFile.
DeviceSession.createOperationId()
Return a new network-mutation operation ID.
DeviceSession.networkStatus(options = {})
Return network.status state/result.
DeviceSession.configureNetwork({ ssid, password, operationId = undefined }, options = {})
Send credentials and operationId; return accepted ACK, not final connection evidence.
DeviceSession.setNetworkEnabled(enabled, options = {})
Send persisted network enable/disable mutation.

`DeviceSession.forgetNetwork(options = {})`
: Request forgetting persisted credentials.

`DeviceSession.release({ timeout = 1200 } = {})`
: Release the EV-MUX session lease; Promise of RPC result.

`DeviceSession.executePython(source, { timeout = null, signal = null } = {})`
: Execute source through framed REPL and return output.

`DeviceSession.interrupt()`
: Send Ctrl-C through the framed signal channel.

`DeviceSession.control(operation, { timeout = 15000, signal = null } = {})`
: Run stop/soft-reset/hard-reset/bootloader control.

`DeviceSession.stopForRepl({ timeout = 5000, signal = null } = {})`
: Interrupt and wait for an executable REPL; returns boolean readiness.

`DeviceSession.runScript(path, content, { timeout = 10000, signal = null, onStarted = null, maxOutputChars = Number.POSITIVE_INFINITY, trial = null, inlineCompletion = false, } = {})`
: Run a device script; optional content uploads first; await bounded completion.

`DeviceSession.startScript(path, content, { onStarted = null, ...options } = {})`
: Start a script and return a handle with done Promise.

`DeviceSession.scriptRunSnapshot()`
: Return the active run snapshot or null.

`DeviceSession.snapshot()`
: Return session state, role, capabilities, parser/transport statistics and pending RPC count.

`DeviceSession.close(error = new Error('connection closed'))`
: Reject pending work and reset session state/timers; physical transport closing is the host responsibility.

Readable negotiated state includes state, role, identity, capabilities, sessionId, vmEpoch, vmReady, limits, parser and repl. Update it through session methods rather than direct assignment. `negotiate` options: role=" unknown" , drainMs=300, acquire=true, probeRepl=true, expectedDevice=null, signal=null. TCP skips startup drain. Serial may complete a crossing partial frame within the parser' s remaining receive budgets while continuing to reject old frames; successive frames cannot extend the initial partial frame' s total deadline. An injected older parser without receive-budget support retains the conservative incomplete-frame failure instead of an unbounded wait. `control` operation strings are stop/soft-reset/hard-reset/bootloader, distinct from the facade request object. Low-level network mutation methods return accepted ACKs; the facade configure waits for matching operation completion. A typed run trial, if used, requires operation_id UUIDv4, source_path (at most 512 UTF-8 bytes), source_hash (64 lowercase SHA-256 hex digits) and the advertised launcher trial/typed-runner features; inlineCompletion cannot be combined with a trial.

EV_MUX_DEFAULT_LIMITS

Fallback protocol limits; actual operations use negotiated firmware limits. txLargePayloadMax=0 is the initial fallback, not a promise that every device supports a zero-byte image only.


```
{
  "metadataMax": 384,
  "rxPayloadMax": 2048,
  "replInputMax": 511,
  "vmPendingMax": 1,
  "txLargePayloadMax": 0,
  "txFrameTimeoutMs": 1500,
  "txQueueTimeoutMs": 2000
}
```


EV_MUX_PATH_MAX_BYTES

Device path UTF-8 byte limit including temporary-name headroom.

```
123
```

EV_MUX_PAYLOAD_LIMIT

Compatibility payload-budget constant; it does not replace the negotiated RX limit.

```
1800
```

EV_MUX_REQUIRED_CHANNELS

Channels required by the v4 capability validator.

```
[
  "user.rpc",
  "debug.rpc",
  "repl.stdin",
  "repl.stdout",
  "repl.stderr",
  "repl.signal",
  "log.idf",
  "preview.frame"
]
```

EV_MUX_REQUIRED_FEATURES

Features required by the v4 capability validator.

```
[
  "capabilities",
  "route.changed",
  "framed.repl",
  "session.firstCome",
  "session.keepalive"
]
```

EV_MUX_SCRIPT_CHUNK_BYTES

Legacy script.write reference constant. SDK 0.3.0 uploads do not use it; binary chunk size comes from fileWriteChunkMax.

```
900
```

EV_MUX_VERSION

Supported session protocol version.

```
4
```

evmuxRpcChannelForMethod

`evmuxRpcChannelForMethod(method)`

Return debug.rpc for debug.* methods and user.rpc otherwise.

requiresEvmuxFirmwareUpgrade

`requiresEvmuxFirmwareUpgrade(err)`

Return boolean for firmware-upgrade errors, including version mismatch or the firmware-update marker.

validateEvmuxCapabilities

`validateEvmuxCapabilities(payload)`

Validate v4 protocol version, required features/channels and the session lease. Returns void; throws on mismatch or missing capabilities.

10.2.5 @esp-vision/device-core/actions

DEVICE_ACTION_DESCRIPTORS

The 17 descriptors below define host action contracts. They do not mean that platform port discovery, connection UI or workspace deployment has been implemented in Core; bind those handlers explicitly. Inputs follow each descriptor inputSchema; outputSchema permits handler-defined extra properties. risk contains readOnly/destructive/openWorld flags.

`device.listPorts`

List platform-native serial candidates for the user to select; this action never chooses a device.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": true,
    "destructive": false,
    "openWorld": false
  },
  "requiresReadyDevice": false,
  "concurrency": "shared-read"
}
```

`device.connect`

Connect to an ESP Vision device using standard elicitation for a serial port or an explicit device TCP address, then acquire an EV-MUX v4 session.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "transport": {
        "enum": [
          "serial",
          "tcp"
        ],
        "default": "serial",
```

(continues on next page)

(continued from previous page)

```
        "description": "Use serial for a platform port or tcp for an IP address on␣
↪the device network."
      },
      "port": {
        "type": "string",
        "description": "Platform-native serial identifier, such as /dev/ttyACM0 or␣
↪/dev/cu.usbmodem*. Omit it only when the MCP host advertises elicitation or ESP_
↪VISION_PORT is configured. Non-interactive callers must pass it explicitly. A␣
↪direct network address may also be supplied as tcp://IP:PORT or IP:PORT;␣
↪alternatively set transport to tcp and provide host and tcpPort."
      },
      "host": {
        "type": "string",
        "description": "IPv4 or IPv6 address of the device when transport is tcp."
      },
      "tcpPort": {
        "type": "integer",
        "minimum": 1,
        "maximum": 65535,
        "default": 3232,
        "description": "Device EV-MUX TCP port."
      },
      "baudRate": {
        "type": "integer",
        "minimum": 1200,
        "maximum": 4000000,
        "default": 115200
      },
      "drainMs": {
        "type": "integer",
        "minimum": 0,
        "maximum": 3000,
        "default": 300
      }
    },
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": false,
    "openWorld": true
  },
  "requiresReadyDevice": false,
  "concurrency": "exclusive"
}
```

`device.disconnect`

Disconnect the active device session and release its lease.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": false
  },
  "outputSchema": {
```

(continues on next page)

(continued from previous page)

```
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": false
  },
  "requiresReadyDevice": false,
  "concurrency": "exclusive"
}
```

`device.negotiate`

Discover EV-MUX v4, explicitly acquire a session, and prove the framed REPL is live.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "role": {
        "type": "string",
        "default": "unknown"
      },
      "drainMs": {
        "type": "integer",
        "minimum": 0,
        "maximum": 3000,
        "default": 300
      }
    },
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": false,
    "openWorld": false
  },
  "requiresReadyDevice": false,
  "concurrency": "exclusive"
}
```

`device.inspect`

Inspect protocol state, device identity, filesystem, and recent logs.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "includeFiles": {
        "type": "boolean",
        "default": true
      },
      "logLimit": {
        "type": "integer",
```

(continues on next page)

(continued from previous page)

```
        "minimum": 0,
        "maximum": 500,
        "default": 100
      }
    },
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": true,
    "destructive": false,
    "openWorld": false
  },
  "requiresReadyDevice": true,
  "concurrency": "shared-read"
}
```

`python.execute`

Execute MicroPython through the shared interpreter using bounded REPL or binary file staging.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "code": {
        "type": "string"
      },
      "timeoutMs": {
        "type": "integer",
        "minimum": 100,
        "maximum": 120000,
        "default": 8000
      }
    },
    "required": [
      "code"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": true
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`python.runEphemeral`

Run one Python draft from the reserved temporary device path without modifying a user-owned file.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "code": {
        "type": "string"
      },
      "maxDurationMs": {
        "type": "integer",
        "minimum": 1000,
        "maximum": 180000,
        "default": 180000
      }
    },
    "required": [
      "code"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": true
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`python.observeEphemeral`

Observe the bounded output and status of the current temporary Python run.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "runId": {
        "type": "string"
      },
      "waitMs": {
        "type": "integer",
        "minimum": 0,
        "maximum": 10000,
        "default": 1000
      },
      "outputOffset": {
        "type": "integer",
        "minimum": 0,
        "maximum": 1048576,
        "default": 0
      }
    },
    "required": [
      "runId"
    ],
    "additionalProperties": false
  },
```

(continues on next page)

(continued from previous page)

```
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": true,
    "destructive": false,
    "openWorld": false
  },
  "requiresReadyDevice": true,
  "concurrency": "shared-read"
}
```

`python.stopEphemeral`

Stop the current temporary Python run and remove its reserved device file.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "runId": {
        "type": "string"
      }
    },
    "required": [
      "runId"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": false
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`deviceFile.read`

Read a file from the device filesystem.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "path": {
        "type": "string"
      },
      "encoding": {
        "enum": [
          "utf-8",
          "base64"
        ],
        "default": "utf-8"
```

(continues on next page)

(continued from previous page)

```
      }
    },
    "required": [
      "path"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": true,
    "destructive": false,
    "openWorld": false
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`deviceFile.write`

Write a file to the device using the negotiated transfer path.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "path": {
        "type": "string"
      },
      "content": {
        "type": "string"
      },
      "encoding": {
        "enum": [
          "utf-8",
          "base64"
        ],
        "default": "utf-8"
      },
      "runAfterWrite": {
        "type": "boolean",
        "default": false
      }
    },
    "required": [
      "path",
      "content"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": true
```

(continues on next page)

(continued from previous page)

```
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`workspaceFile.deploy`

Stream one workspace-relative file to the device without placing binary content in the model context.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "sourcePath": {
        "type": "string",
        "description": "Workspace-relative source file path."
      },
      "devicePath": {
        "type": "string",
        "description": "Absolute destination path on the device."
      },
      "expectedBytes": {
        "type": "integer",
        "minimum": 0,
        "maximum": 16777216
      },
      "expectedSha256": {
        "type": "string",
        "description": "Optional lowercase SHA-256 expected before transfer."
      },
      "overwrite": {
        "type": "boolean",
        "default": false
      }
    },
    "required": [
      "sourcePath",
      "devicePath"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": true
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`deviceFile.remove`

Remove one file or one empty directory from the device filesystem.

```
{
  "inputSchema": {
    "type": "object",
```

(continues on next page)

(continued from previous page)

```
    "properties": {
      "path": {
        "type": "string"
      },
      "kind": {
        "enum": [
          "file",
          "directory"
        ],
        "default": "file"
      }
    },
    "required": [
      "path"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": true
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`deviceTest.run`

Write, run, and capture a target-side MicroPython test.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "path": {
        "type": "string",
        "default": "/.agent_test.py"
      },
      "source": {
        "type": "string"
      },
      "timeoutMs": {
        "type": "integer",
        "minimum": 500,
        "maximum": 120000,
        "default": 10000
      }
    },
    "required": [
      "source"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
```

(continues on next page)

(continued from previous page)

```
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": true
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`device.control`

Interrupt, reset, or enter the bootloader.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "operation": {
        "enum": [
          "stop",
          "soft-reset",
          "hard-reset",
          "bootloader"
        ]
      },
      "confirmBootloader": {
        "type": "boolean",
        "default": false
      }
    },
    "required": [
      "operation"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": false
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`device.rpc`

Call a negotiated user or debug RPC method. For network.configure, network.enable, and network.forget, include a 32-character lowercase hexadecimal operationId and reuse the exact same ID and payload when retrying after a timeout.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "method": {
        "type": "string"
```

(continues on next page)

(continued from previous page)

```
      },
      "payload": {
        "type": "object",
        "default": {}
      },
      "channel": {
        "enum": [
          "user.rpc",
          "debug.rpc"
        ]
      },
      "timeoutMs": {
        "type": "integer",
        "minimum": 100,
        "maximum": 120000,
        "default": 4000
      }
    },
    "required": [
      "method"
    ],
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": false,
    "destructive": true,
    "openWorld": true
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

`frame.capture`

Capture one JPEG frame for an explicitly requested observation.

```
{
  "inputSchema": {
    "type": "object",
    "properties": {
      "timeoutMs": {
        "type": "integer",
        "minimum": 500,
        "maximum": 30000,
        "default": 8000
      }
    },
    "additionalProperties": false
  },
  "outputSchema": {
    "type": "object",
    "properties": {},
    "additionalProperties": true
  },
  "risk": {
    "readOnly": true,
    "destructive": false,
```

(continues on next page)

(continued from previous page)

```
    "openWorld": false
  },
  "requiresReadyDevice": true,
  "concurrency": "exclusive"
}
```

DEVICE_ACTION_SCHEMA_VERSION

Schema version of host action descriptors.

```
1
```

DEVICE_ACTORS

Allowed actor names for the host action layer; the ordinary device facade does not require an actor.

```
[
  "agent"
]
```

DeviceActionError

```
new DeviceActionError(code, message, { recovery = null, details = null, cause
= null } = {})
```

Structured host-action error; code/message plus optional recovery/details/cause. toJSON omits cause.

DeviceActionError.toJSON()

Return {code,message,recovery?,details?}.

DeviceActionGateway

```
new DeviceActionGateway({ registry = new DeviceActionRegistry(), coordinator
= new OperationCoordinator(), lease = new DeviceLease(), getSession = () =>
null, authorize = async () => true, } = {})
```

Validates action input and context, checks the host lease/ready session, calls authorize(descriptor, input, context), then coordinates the bound handler. context.actor must be supported and context.requestId must be supplied; context.signal supports cancellation. Host code must supply its actual authorization policy; the default authorize returns true.

DeviceActionGateway.invoke(actionName, input = {}, context = {})

Promise: validated/authorized bound-handler result; errors carry action context.

DeviceActionRegistry

```
new DeviceActionRegistry(descriptors = DEVICE_ACTION_DESCRIPTORS, {
schemaVersion = DEVICE_ACTION_SCHEMA_VERSION } = {})
```

Registry of action descriptors and explicitly bound handlers. Descriptors require name, inputSchema, outputSchema and risk. Unbound descriptors are metadata; Gateway.invoke rejects them with ACTION_UNAVAILABLE. register/bind return this for chaining.

DeviceActionRegistry.register(descriptor, handler = null)

Register a descriptor and optional handler.

DeviceActionRegistry.bind(name, handler)
Bind a handler to an existing action.
DeviceActionRegistry.descriptor(name)
Return descriptor or null.
DeviceActionRegistry.handler(name)
Return handler or null.
DeviceActionRegistry.list()
Return descriptor array.
DeviceActionRegistry.validate(name, input)
Validate/default input and return the validated copy.

DeviceLease

```
new DeviceLease()
```

Host-side actor lease, distinct from the firmware EV-MUX session lease. actor must be in DEVICE_ACTORS. Multiple handles for the same actor are reference counted; this object neither acquires nor forcibly steals a device session.

DeviceLease.acquire(actor)
Return {id,actor,release}; a conflicting actor throws.
DeviceLease.release(id)
Release one host lease ID.
DeviceLease.assertAccess(actor)
Throw unless actor owns the host lease.
DeviceLease.snapshot()
Return {state,primary,actors,holders}.

DeviceLeaseConflictError

```
new DeviceLeaseConflictError(owner, requester)
```

DeviceActionError for a host actor conflicting with the current lease owner.

OperationCoordinator

```
new OperationCoordinator()
```

Serial host operation queue with cancellation and close handling. Unlike the device facade, this adapter layer queues operations. operation receives its AbortSignal.

OperationCoordinator.run({ actor, requestId, actionName, signal }, operation)
Enqueue operation and return its result Promise.
OperationCoordinator.close(reason = actionAbortError('Device session closed'))
Reject queued work and abort active work; no new operations accepted.
OperationCoordinator.snapshot()
Return active and queued actor/request/action identities.

10.2.6 @esp-vision/device-core/bindings

DeviceEvidenceBuffer

```
new DeviceEvidenceBuffer(limit = MAX_EVIDENCE_ENTRIES)
```

Bounded host evidence history. Default constructor limit is 500; snapshot defaults to 100. Values are bounded; applications remain responsible for sensitive-data handling.

DeviceEvidenceBuffer.append(kind, value, details = null)
Append a timestamped bounded value and optional details.
DeviceEvidenceBuffer.snapshot(limit = 100)
Return the latest limited entries.
DeviceEvidenceBuffer.clear()
Clear history.

EPHEMERAL_PYTHON_PATH

Reserved path for the advanced temporary-run controller; ordinary source/start uses its own unique path.

```
"/.espvision_agent_run.py"
```

EphemeralPythonController

`new EphemeralPythonController(session, { evidence = null, generation = 'session' } = {})`

One bounded temporary Python run per controller, using EPHEMERAL_PYTHON_PATH. Constructor session is a DeviceSession. Snapshots include runId/path/status/output/nextOffset/finished/cleaned and optional error/outputTruncated/cleanupError/residualPath; a started run is not proof that cleanup succeeded.

EphemeralPythonController.start(code, { maxDurationMs = 180000, signal = null } = {})
Start bounded temporary source; return the run snapshot and clean up after completion.
EphemeralPythonController.observe(runId, { waitMs = 1000, outputOffset = 0, signal = null } = {})
Wait up to waitMs and return incremental output from outputOffset.
EphemeralPythonController.stop(runId)
Stop the identified run and return its cleanup/status snapshot.
EphemeralPythonController.close()
Stop/clean up the active run when disposing the controller.

bindSessionDeviceActions

`bindSessionDeviceActions(registry, { evidence = new DeviceEvidenceBuffer(), lease = null, getOperations = () => null, getGeneration = () => 'session', } = {})`

Bind generic session/file/Python/capture handlers to a registry and return { evidence, controllerFor }. Host-specific listPorts/connect/disconnect/workspace deployment are not provided by this binding and must be implemented by the host.

10.2.7 @esp-vision/device-core/firmware

Addressed ROM operations

The public `openNodeRom` and `openBrowserRom` adapters also support callers that manage their own Flash layout. After `initialize()`, qualify the selected chip/revision and `getSecurityInfo()` before `runStub()`; operate only after the stub is running and call `disconnect()` in cleanup. These operations are addressed, not restricted to a FAT partition name:

- `stub.eraseRegion(address, size)` erases the supplied range. Address and positive size must be 4096-byte aligned and fit the 32-bit address space.
- `stub.flashData(arrayBuffer, onProgress, offset)` writes the supplied bytes at the caller-selected offset through the official flasher.

- `stub.readFlash(offset, byteCount, onProgress?, options?)` reads the caller-selected range. `options` accepts `chunkSize`, `blockSize` (1024 or 4096) and `maxInFlight`; large verification reads should loop over at most 65536 bytes with a 1024-byte packet and one outstanding packet.
- `stub.flashId()` returns the integer JEDEC ID, with the capacity exponent in bits 16–23, not a capacity in bytes. Callers must check that their selected ranges fit the actual chip. Callers also own partition selection, overlap decisions, data hashes and readback verification.

For portable Node/browser use, plan one `flashData()` call per adapter instance. The browser adapter rejects a second call with `ROM_WRITE_REUSE` and closes the session. Merge multiple segments into one continuous payload with intentional contents for the gaps, or establish and qualify a new ROM session for each write. Do not replay a write after an uncertain result.

These primitives are the flexible layer for custom partition tools. `FirmwareFlashSession` adds the approved-image installation/update workflow, complete verification and target-specific reset handling. Core 0.7.0 adds the previously missing cross-platform region-erase primitive and changes complete-installation deletion semantics. It does not add a fixed FAT address: that workflow parses the approved partition table, so moving a partition does not itself require a new Core release. Application-only updates keep user files.

`stub.flashData()` may return `{ algorithm: 'md5', offset, bytes, digest }` only after official esptool successfully compares the device digest for that exact write range. The digest is 32 lowercase hexadecimal characters. Core rejects malformed, mismatched-range or mismatched-digest receipts without reset or retry. Returning null/undefined retains legacy full readback. SHA-256 validation of the downloaded source and embedded install segments remains mandatory; MD5 detects programming errors and is not a cryptographic replacement for SHA-256.

FirmwareFlashSession

```
new FirmwareFlashSession(loader, image, { reset = () => loader.hardReset(),
mode = 'update', signal? } = {})
```

ROM-loader adapter, separate from a live EV-MUX session. Each instance has one lifetime. state progresses idle → connecting → connected → flashing → verified → closed, or error → closed. flash always disconnects, so closed alone is not evidence that verification succeeded.

`FirmwareFlashSession.connect(bytes)`
Validate image, chip/revision/security mode and installed partition-table hash before accepting writes.

`FirmwareFlashSession.flash(bytes, { onProgress = () => {}, onPhase = () => {} } = {})`
Write and verify the application partition, then reset; always disconnect. Built-in adapters return the device MD5 receipt checked by official esptool; Core verifies its exact offset, length and digest against the SHA-256-qualified source. Custom adapters without a receipt retain complete SHA-256 readback. Progress is 0..100 and onPhase receives status strings.

`FirmwareFlashSession.disconnect()`
Close loader once; subsequent calls are no-ops.

Image descriptor: evMuxVersion=4, bytes, offset, partitionSize, sha256, chipFamily, chipRevision and partitionTable:{offset,bytes,sha256}. Supply trusted build metadata and an exact Uint8Array; default mode never performs whole-chip erase or partition migration. The injected loader provides initialize(), chipFamily/chipRevision, getSecurityInfo(), runStub(true), hardReset(), disconnect(). The returned stub provides flashData(), readFlash(), readRegister() and writeRegister(). P4/S31 normal-boot latch handling is built in, using each target's register address. A reset failure after verified writing is reported through onPhase; it does not undo verification or automatically flash again.

`mode: 'install'` uses the approved complete image and derives the writable, unencrypted `ffat` range from its partition table. Chip/revision/security, JEDEC capacity, bounds and partition integrity checks remain mandatory. The stub provides `flashId()`, `eraseRegion(address, size)`, `flashData()` and `readFlash()`. The sector-aligned FFAT range is erased before writing; a successful erase command is sufficient, with no erased-range readback, sampling or digest calculation. Erase failure or cancellation before writing rejects without reset or replay. Invalid layouts return FIRMWARE_PARTITION_INVALID or FIRMWARE_FATFS_UNAVAILABLE; missing capabilities return FIRMWARE_ERASE_UNSUPPORTED. No whole-chip erase, backup or automatic restore is performed. See *Application API reference* for deletion semantics.

`firmwareMode(mode = 'update')` validates and returns update/install; other values throw TypeError. `validateFirmwareImage(image, bytes, { mode = 'update' } = {})` also validates embedded bootloader/partition/application ranges and hashes for install mode.

firmwareSha256

`firmwareSha256(bytes)`

Promise<string>: lowercase SHA-256 hexadecimal digest; requires Web Crypto.

validateFirmwareImage

`validateFirmwareImage(image, bytes, { mode = 'update' } = {})`

Promise<Uint8Array>: validate v4, exact byte size, application partition bounds and SHA-256; returns the supplied bytes. It does not infer or download an approved manifest.

10.2.8 @esp-vision/device-core/node

`listSerial({ signal, timeout = 10000 } = {}, dependencies = {})`
: Passively enumerate OS serial ports through npm native bindings; no Python or port opening.

`filterSerialPorts(ports, { platform = process.platform, allPorts = false } = {})`
: Filter OS-enumerated records by platform path and USB descriptor hints without opening ports or claiming firmware identity.

`serialEntry(port)`
: Convert one serial descriptor into an unprobed interface/address candidate with USB metadata; descriptor classification is not device identity.

`transportFor(entry, options = {})`
: Create a passive TCP or serial byte transport for an inventory entry.

`inventory(options = {}, dependencies = {})`
: Return `{ devices, warnings, sourcesAvailable }` from serial enumeration and LAN discovery. Defaults to a passive scan; `probe: true` connects to each candidate. Use `timeout`, `interfaces`, `allPorts` and `signal` to control the scan.

Passive native USB ROM entries return `status: rom` and `identity: null` without opening or resetting the device. With `probe: true`, Core verifies the ROM factory MAC and security mode, clears the target-specific software download latch, then requests normal SPI boot once. It does not load a stub, erase or read Flash. Only a unique endpoint matching both factory MAC and physical USB location may receive the subsequent runtime handshake. `romProbe` records `attempted`, `outcome` (`runtime`, `rom` or `no-runtime`), selected MAC/location and reset/cleanup diagnostics. A device already returned to runtime is probed without another reset. No returned runtime remains `unidentified` with ROM_RUNTIME_NOT_FOUND; a returned ROM remains `rom`. Neither outcome proves installed firmware presence or absence. An unbootable application may require physical BOOT/RESET before installation. Only a successful ESP-VISION handshake establishes firmware identity; serial/session ownership conflicts return `busy`.

Core 0.8 removes `defaultPython`, `nodeAdapterAssets` and the Python helper files. Bundlers must preserve npm native loaders and prebuilds; see the CLI build script.

`tcpTransport({ host, port })`
: Return a passive Node TCP byte adapter. Core opens it during `device.connect({ role: 'tcp' })`.

`serialTransport({ serial })`
: Return the npm native serial byte adapter. Use an explicit path and role `usj`, `usb` or `uart`. Core opens/closes it and owns the device session.

`openNodeRom({ port, signal?, logger?, baudRate?, onStatus? })`
: Return the built-in ROM adapter without opening the port. Use with `device.firmware.update({ openLoader: ({ signal }) => openNodeRom({ port, signal }), ... })` for both

installation and application updates. The adapter shares the browser esptool-js engine and requires no host Python. Core manages initialize, validation, writing/verification, reset and disconnect. Available from SDK 0.3.0; it is Node-only, not a runtime serialTransport or a TCP OTA adapter.

10.2.9 @esp-vision/device-core/discovery

ESP_VISION_DISCOVERY_SERVICE

DNS-SD service queried by the Node provider.

```
"_esp-vision._tcp.local"
```

NetworkDiscovery

```
new NetworkDiscovery(provider)
```

Normalizes a provider with list({timeout, signal}), validates results and sorts by hostname. list uses the same limits/result shape as device.discover; construction does not start a scan.

`NetworkDiscovery.list({ timeout = 3000, signal } = {})`
Collect normalized records within the discovery timeout; supports AbortSignal.

createBridgeDiscovery

```
createBridgeDiscovery(connectionUrl, { fetch: fetcher = globalThis.fetch }
= {})
```

Return NetworkDiscovery backed by the bridge GET /devices?timeout=⋯endpoint. Accept ws/wss connectionUrl and optional fetch injection. Returned connectionUrl must match the bridge origin and a /device/<32-hex> path.

normalizeDiscoveredDevice

```
normalizeDiscoveredDevice(value)
```

Return a normalized record or throw: 12 lowercase MAC hex digits, DNS hostname ≤63 characters, board ≤96, firmwareVersion ≤255, integer evMuxVersion, port 1..65535 and at least one valid IPv4. Deduplicate addresses; compatible means v4.

10.2.10 @esp-vision/device-core/discovery/node

createNodeDiscovery

```
createNodeDiscovery({ createMdns, interfaceAddress } = {})
```

Return NetworkDiscovery using DNS-SD/mDNS PTR/SRV/TXT/A records for _esp-vision._tcp.local. createMdns optionally injects a multicast-dns-compatible factory; interfaceAddress selects an interface IPv4. Otherwise multicast-dns is dynamically imported. Accepts private/link-local IPv4 candidates, honors TTL and destroys sockets after each scan.

10.2.11 Integration boundaries

Subpath exports are executable implementations and host contracts, not aliases of every facade request. Keep one session coordinator per physical device. Device owners, host actor leases, ROM loader lifetimes and discovery scans have different meanings; use the facade for ordinary device work and an explicit subpath only when implementing that host layer.

10.2.12 FirmwareFlashSession.flash result

After verified writing, flash returns `{ verified: true, needsReset: boolean }`. Reset failure sets needsReset; closed state alone is not verification evidence. Application integrations normally use `device.firmware.update()` to include release checks and port handoff.

10.2.13 @esp-vision/device-core/browser

`openBrowserRom({ port, logger = {}, signal?, expectedMac? })`

Included in SDK 0.6.0. Return a passive browser ROM adapter for a selected Web Serial `SerialPort`. Request the port in the user gesture and release any runtime connection first. `initialize()` lazily loads official `esptool-js@0.7.1`, opens/detects the chip and exposes `chipName/chipFamily/chipRevision`; it does not start a stub. The adapter implements `initialize/getSecurityInfo/runStub/hardReset/disconnect` for `firmware.update`. Core checks the approved image and ROM security before starting the stub and owns all writing/verification/reset semantics. `logger` optionally provides `log/error/debug` functions. `signal` cancels initialization and region erase. Core also checks cancellation after erase and before firmware writing; once writing starts, verification/reset finishes as before. `disconnect()` is idempotent and releases the port; initialization failures also close acquired resources. A disconnected adapter cannot be reused. Use a browser bundler; direct upstream native ESM loading is unsupported. See *Integrating the Device Core npm Package* for the complete example. This subpath imports no Node modules; Node uses the same ROM engine through its own serial adapter.

Core 0.7.0 also exposes `chipMac: null` before qualification, then the factory MAC as 12 lowercase hexadecimal characters, matching the Node adapter. The pinned official chip `readMac(loader)` primitive supplies the value. Optional `expectedMac` accepts exactly 12 hexadecimal characters (uppercase is normalized); separators, whitespace, `null` and non-string values synchronously reject before port access. The host must obtain the selected runtime's verified factory MAC, for example a validated `machine.unique_id()` known to match that board's factory MAC; arbitrary legacy `deviceId` values or USB VID/PID do not establish this association. If the value cannot be verified, use an explicitly user-selected manual ROM workflow rather than infer an automatic association. A first mismatch or recovery to a different family/revision/MAC returns ROM_IDENTITY_CHANGED, closes the adapter and prevents another stub, erase or write; malformed MAC replies return ROM_IDENTITY_INVALID. These codes are retained on USB-OTG rather than being replaced with manual-download guidance. Omitting the option preserves manual initial ROM selection, but subsequent synchronization remains bound to that first device.

10.2.14 Release readers in @esp-vision/device-core/firmware (0.5.1)

`loadOfficialFirmware({ board?, version?, fetch = globalThis.fetch, signal? })`

Load the official current or exact selected version snapshot through the publication reference; return the manifest with publication history. Validate the entire snapshot before optional board filtering. Configs without publication retain legacy metadata compatibility.

`loadReleaseManifest({ url, sha256, version?, sourceCommit?, boards?, fetch = globalThis.fetch, signal? })`

Load exactly one immutable official manifest, validate raw-byte SHA-256, identities, complete board set, every firmware descriptor and website/model/MCP source boundaries. Return the manifest object without downloading firmware or opening ROM. Limits, fields and single-BIN update behavior are specified in *Application API reference*.

10.2.15 Browser flash startup

The browser ROM adapter resets through the active flash stub after verification. A verified write does not prove that firmware has started: hosts must wait for a fresh runtime handshake before enabling configuration. If `needsReset` is true, `resetError` carries the reset failure code and message; preserve this diagnostic and offer manual reset only as recovery.

Node ROM readback safety

Node and browser Flash reads use one outstanding 1024-byte packet and verify each bounded command's trailing MD5. Node retains large-read support by splitting requests into at most 64 KiB commands; legacy packet/window options are upper budgets. Partition-table validation uses this same path before writing. A short read, timeout or digest error fails without replay.

If a read-only initialization fails while a previously observed native ROM device is re-enumerating, Core passively waits for up to five seconds. Only the unique runtime with the same factory MAC and physical USB location may receive one verified bootloader request. Ambiguous or conflicting identities cannot trigger a handoff. This recovery never repeats a submitted bootloader request or a Flash read/write.

10.3 Integrating the Device Core npm Package

For the SDK overview and complete API index, see *device-core/index*; application parameters/results/events are in *device-core/api* and all advanced exports in *device-core/advanced*. This page retains runnable integration examples.

This page targets host developers connecting an IDE, Device CLI, agent workflow or custom host application to a device. `@esp-vision/device-core` is the shared UI-independent library. Hosts supply physical byte connections without copying the reference Python client or implementing another protocol stack. Agent device operations use the CLI; Knowledge MCP serves documentation over HTTP. See *communication/wire-protocol* for framing and *communication/interfaces* for physical interfaces.

10.3.1 Source, versions and installation

Device Core is maintained in `host/device-core/` in the ESP-VISION firmware repository. `host/` contains code running on the host, not firmware modules. IDEs and custom applications consume the SDK directly; agent workflows use the separately packaged Device CLI for device control. Protocol changes should align firmware, Core, documentation and tests in one MR.

The package has its own version, independent of firmware and IDE versions. These examples target `@esp-vision/device-core@0.6.0` and EV-MUX v4 only. v3 returns `VERSION_MISMATCH` and applications should request a firmware update. Declare a minor-locked range, record the resolved patch in the lockfile, and verify it in the public npm registry before installing; a dry run or an open MR alone does not publish it.

```
npm view @esp-vision/device-core@0.6.0 version dist.integrity
npm install '@esp-vision/device-core@~0.6.0'
```

Node.js 22 is the Node host baseline. Browsers need a bundler/import map for package specifiers and Web Serial or a binary WebSocket bridge. Device Core contains no serial driver, network bridge, model or IDE UI. Node serial enumeration uses optional npm native bindings; Node mDNS discovery uses `multicast-dns`. The separate Device CLI installs as `@esp-vision/device-cli` and provides the supported agent-facing device command. Pin npm versions and update lockfiles.

10.3.2 Application, Core and transport responsibilities

The same SVG is used by the project README and system architecture chapter. Host applications select devices and provide UI/workflow policy; Device Core owns the shared API, protocol and session state; EV-MUX defines runtime communication with the firmware. Byte adapters supply serial/socket I/O without parsing another protocol stack. Representative APIs in the figure do not replace the full API reference. Discovery and ROM flashing have separate lifecycles; the ROM loader does not carry image writes through EV-MUX.

Create one `createDevice({ transport })` object per device connection. The package root exports only this factory. Core owns parsing, request correlation, sessions, keepalive, REPL coordination and operation lifetimes.

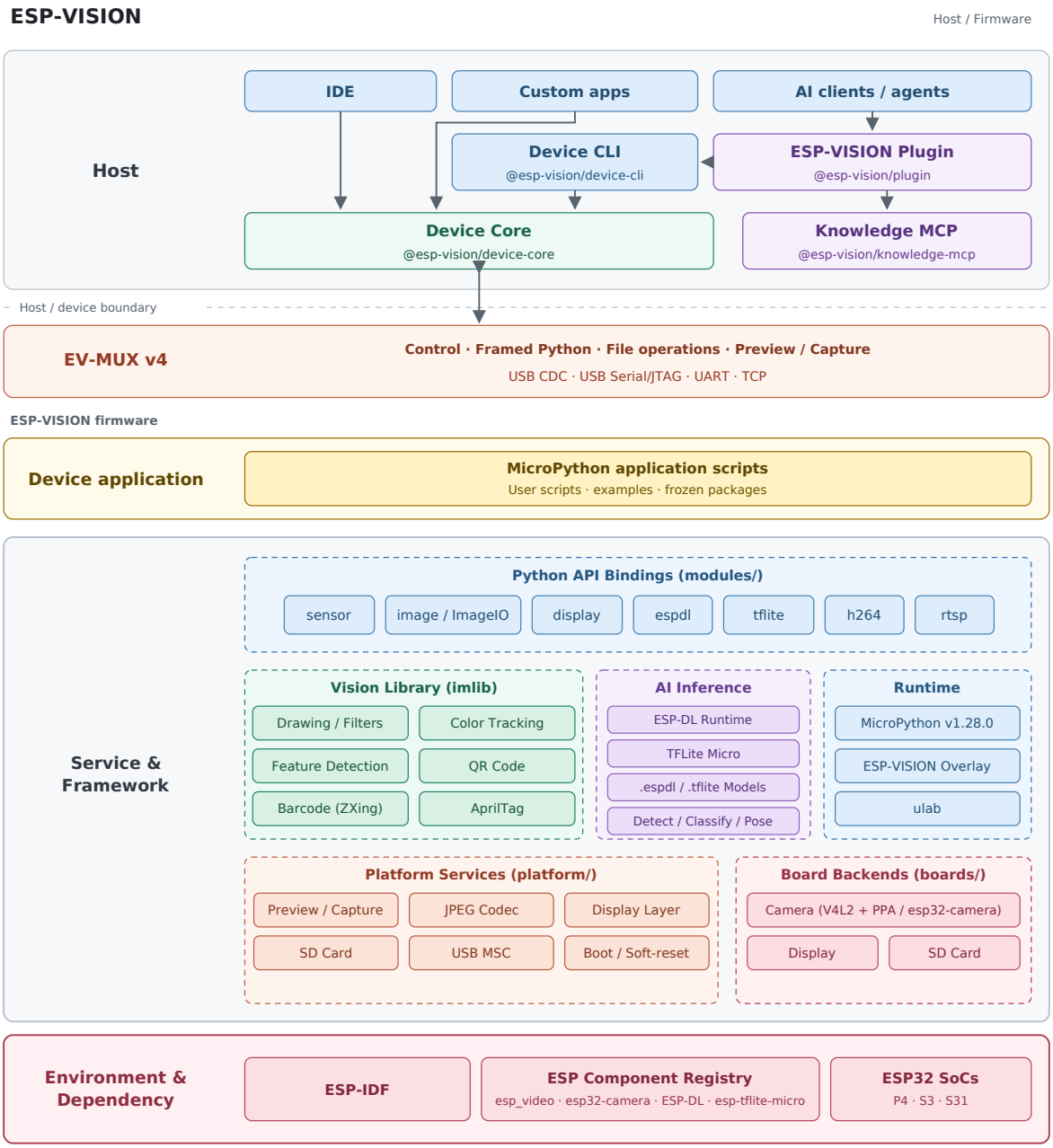


Fig. 1: IDE / Device CLI / custom host apps → Device Core → EV-MUX → firmware; Knowledge MCP → documentation over HTTP

Applications own selection, presentation and policy. Transport adapters own physical connections and reliable raw bytes; bridges must neither acquire ownership nor re-encode EV-MUX.

The transport contract has two required methods. `writeBytes(Uint8Array): Promise` resolves after all bytes are accepted by the host connection and rejects failures. `subscribe({ data, error, close })` forwards raw bytes, errors and closure and must synchronously return an unsubscribe function. Optional `connect({ signal })` / `disconnect()` open and close the physical link. Core subscribes before connecting or handshaking; omit `connect` for an already open link. Short writes and WOULD_BLOCK are not completed writes. Unsubscribed connections must stop delivering data.

10.3.3 Standalone Node TCP example

Save the following as `core-tcp.mjs` in a project with the package installed and run `node core-tcp.mjs <device-ip>`. The device must be online with v4 firmware; its default TCP port is 3232. This example uses a supplied address; discovery is described below.

```
import net from 'node:net'
import { once } from 'node:events'
import { createDevice } from '@esp-vision/device-core'

const host = process.argv[2]
if (!host) throw new Error('Usage: node core-tcp.mjs <device-ip>')
const socket = new net.Socket()
const transport = {
  subscribe({ data, error, close }) {
    socket.on('data', data)
    socket.on('error', error)
    socket.on('close', close)
    return () => {
      socket.off('data', data)
      socket.off('error', error)
      socket.off('close', close)
    }
  },
  async connect({ signal }) {
    socket.connect({ host, port: 3232 })
    await once(socket, 'connect', { signal })
  },
  writeBytes(bytes) {
    return new Promise((resolve, reject) => {
      socket.write(bytes, error => error ? reject(error) : resolve())
    })
  },
  async disconnect() { socket.destroy() },
}
const device = createDevice({ transport })
const unsubscribe = device.on('python.output', text => process.stdout.write(text))
try {
  await device.connect({ role: 'tcp' })
  console.log(await device.inspect({ fields: ['identity', 'capabilities'] }))
  console.log(await device.python.execute({ source: 'print(1 + 2)', completion:
↪'wait' }))
} finally {
  try { await device.disconnect() }
  finally { unsubscribe() }
}
```

`connect()` negotiates ownership and probes the REPL, potentially interrupting existing Python programs. It is not a passive availability check. `inspect()` defaults to identity, capabilities and runtime; use `fields` to request `network` / `storage` explicitly. `disconnect()` releases ownership, ends pending work, unsubscribes and calls the transport close method. Always clean up in `finally`, including after connection failure.

10.3.4 Browser Web Serial example

Call `connectSerial()` from a user click; `requestPort` requires a secure context and user selection. This example uses `role: 'usb'` for USB CDC; use `'usj'` for USJ. Release reader/writer locks before closing the port.

```
import { createDevice } from '@esp-vision/device-core'

async function connectSerial() {
  const port = await navigator.serial.requestPort()
  await port.open({ baudRate: 115200 })
  const reader = port.readable.getReader()
  const writer = port.writable.getWriter()
  let listeners = null
  let reading = null
  const transport = {
    subscribe(next) {
      listeners = next
      reading = (async () => {
        try {
          for (;;) {
            const { value, done } = await reader.read()
            if (done) break
            listeners?.data(value)
          }
          listeners?.close()
        } catch (error) { listeners?.error(error) }
        finally { reader.releaseLock() }
      })()
      return () => { listeners = null }
    },
    writeBytes: bytes => writer.write(bytes),
    async disconnect() {
      await reader.cancel().catch(() => {})
      await reading
      writer.releaseLock()
      await port.close()
    },
  }
  const device = createDevice({ transport })
  const unsubscribe = device.on('python.output', text => console.log(text))
  try {
    await device.connect({ role: 'usb' })
    await device.python.execute({ source: 'print(123)', completion: 'wait' })
  } finally {
    try { await device.disconnect() }
    finally { unsubscribe() }
  }
}
```

Browser networking uses a binary WebSocket bridge with the same transport contract and `role: 'tcp'`. Adapters must check WebSocket OPEN, bound `bufferedAmount` and report disconnection; calling `send` does not prove device receipt. UIs should subscribe to `state` and handle unexpected closure. See *communication/host-integration* for bridging.

10.3.5 Files and interactive Python

These operations assume a connected `device`. Receive output through `device.on('python.output', handler)`. Upload and execution are separate; there is no implicit run-after-upload option.

```
const source = 'print("device-core example")\n'
await device.files.write({ path: '/core-example.py', data: source })
const readback = await device.files.read({ path: '/core-example.py', encoding:
↪'utf-8' })
if (readback !== source) throw new Error('Readback mismatch')
const run = await device.python.execute({ path: '/core-example.py', completion:
↪'start' })
const result = await run.done
console.log(result.ok, result.path)
await device.files.remove({ path: '/core-example.py' })

await device.python.input('print(123)\r')
await device.control({ action: 'stop' })
```

`python.execute` accepts exactly one of `source` and `path`. `completion: 'wait'` awaits the result; `'start'` returns a handle with a `done` Promise. Upload success and startup do not prove execution completion. Infinite programs need `control({ action: 'stop' })`, continued receive processing and an execution result; a stop ACK does not prove resource cleanup.

`files.list/read/write/remove/mkdir` are the sole application file API. `files.write` accepts strings or `Uint8Array`; SDK 0.3.0 sends both through `file.write` raw binary blocks with continuous native writes and SHA-256 commit. Firmware must advertise `file.write.binary`; unsupported versions require an upgrade. Reads and directory operations currently use framed REPL. The filesystem must be mounted and writable with temporary-file space. Terminal input, automated scripts and filesystem helpers share one interpreter. Core rejects conflicting automated operations with `DEVICE_BUSY`; applications must not inject terminal input into a REPL held by an automated operation. See *communication/python-files* for details.

10.3.6 Wi-Fi configuration and status

```
await device.configure({ network: {
  credentials: { ssid: '<wifi-ssid>', password: '<wifi-password>' },
  enabled: true,
}, timeout: 45000 })
const { network } = await device.inspect({ fields: ['network'] })
console.log(network.state, network.ipv4)
await device.configure({ network: { enabled: false } })
// Clear saved credentials and disable networking:
// await device.configure({ network: { credentials: null } })
```

`configure` applies partial updates: omitted fields stay unchanged. `credentials: null` clears saved credentials and disables networking; it cannot accompany `enabled: true`. Credentials require both `ssid` and `password`; use an empty password for an open network. Core correlates the transaction and waits for completion rather than returning only firmware acceptance. Combined changes are sequential, not atomic. On failure, `error.completed` / `error.pending` describe acknowledged changes and unfinished operations; Core does not roll back. Pass `timeout` and `signal` in the same request object.

Prefer USB for provisioning because network changes may disconnect TCP. Do not log credentials; queries do not return saved passwords. Release USB before establishing a TCP session rather than transferring its token.

10.3.7 Discovery and identity checks

Firmware advertises `_esp-vision._tcp.local` with main-chip device ID, full board name, firmware version, canonical hostname and protocol version. The hostname is `esp-vision-<full-board-name>-<last-6-mac-hex>-<firmware-version>` and is consistent across serial, TCP, Python and discovery queries. Discovery does not connect to a device or acquire ownership; online does not mean idle.

Install the optional `multicast-dns` peer for Node discovery:

```
import { createDevice } from '@esp-vision/device-core'
import { createNodeDiscovery } from '@esp-vision/device-core/discovery/node'
import { tcpTransport } from '@esp-vision/device-core/node'

const device = createDevice({ discovery: createNodeDiscovery() })
const devices = await device.discover({ timeout: 3000 })
const selected = devices.find(item => item.compatible && item.addresses.length)
if (!selected) throw new Error('No compatible device found')
const tcpAdapter = tcpTransport({ host: selected.addresses[0], port: selected.port␣
↪})
try {
    await device.connect({ transport: tcpAdapter, role: 'tcp', expectedDevice:␣
↪selected })
    console.log(await device.inspect({ fields: ['identity'] }))
} finally { await device.disconnect() }
```

Applications call `device.discover()`. `createNodeDiscovery()` and `createBridgeDiscovery()` return providers with `list({ timeout, signal })`, not `discover()`; inject them through `createDevice({ discovery: provider })`. SDK 0.8 exports `inventory`, `listSerial`, `filterSerialPorts`, `serialEntry`, `transportFor` from `@esp-vision/device-core/node`. Inventory is passive unless `probe: true`; transport constructors do not open links. Serial enumeration requires npm native prebuilds, without host Python. See *Advanced export API reference*.

```
import { inventory } from '@esp-vision/device-core/node'

const { devices, warnings } = await inventory({ timeout: 3000 })
console.log(devices.map(({ interface, address, status }) => ({ interface, address,␣
↪status })))
console.warn(warnings)
```

The Node inventory combines OS serial enumeration and LAN discovery and returns `devices`, `warnings` and `sourcesAvailable`. By default it does not open candidates; `probe: true` connects to each one and may interrupt running Python. Use *Advanced export API reference* for option defaults, candidate filtering and bundler asset handling.

Browsers inject `createBridgeDiscovery('ws://127.0.0.1:8766/device')` from `@esp-vision/device-core/discovery` using a local bridge. Browsers cannot natively enumerate DNS-SD. Results contain `deviceId`, `hostname`, `board`, `firmwareVersion`, `addresses`, `port` and `compatible`; bridge results may also contain `connectionUrl`. The host builds a transport from the selection and passes `expectedDevice` to match ID and hostname before acquisition, protecting against stale DHCP addresses. This is not authentication. VLANs, multicast filtering and AP isolation may require manual addresses; refresh expired bridge handles.

10.3.8 Preview, logs and capture

```
const unsubscribePreview = device.on('preview', image => {
  const url = URL.createObjectURL(new Blob([image.data], { type: image.mimeType }))
  const previous = imageView.src
  imageView.src = url
  if (previous.startsWith('blob:')) URL.revokeObjectURL(previous)
})
const unsubscribeLog = device.on('log', text => systemLog.write(text))
const image = await device.capture({ source: 'camera', format: 'jpeg', timeout:␣
↪15000 })
// image: { data: Uint8Array, mimeType: 'image/jpeg', metadata }
// Call unsubscribePreview()/unsubscribeLog() and revoke the final URL on UI␣
↪teardown.
```

`imageView` and `systemLog` are host UI objects. `on` returns an unsubscribe function. Image events and capture results contain raw JPEG bytes; UIs do not parse EV-MUX again. Core checks session and epoch before dis-

patch. Device programs trigger preview using `img.flush()` and frames may be dropped under congestion. Camera observation needs an executable VM and both `debug.capture_frame` and `debug.capture_frame.observed.v1`. It only encodes the independent frame published by the application's last successful snapshot; it never initializes or physically captures from the sensor and never overwrites an application-held image. Missing safety support rejects before sending the RPC with `UNSUPPORTED_OPERATION` and an upgrade message. `CAMERA_FRAME_NOT_READY` means no valid published frame; `NO_MEM` reports failure to allocate the observation copy without failing the application snapshot. Connect with `probeRepl: false` to avoid interrupting an existing application.

`capture({ source: 'screen', format: 'jpeg' })` calls dedicated `debug.capture_screen` and returns JPEG encoded from the last submitted LCD RGB565 frame. It runs in the VM task without camera calls or screen changes. An absent LCD returns `NOT_SUPPORTED`; an uninitialized display or absent valid frame returns `DISPLAY_NOT_READY`. It never falls back to the camera. Core reports unsupported capability on older firmware. High-throughput raw binary upload remains unimplemented. Hardware testing uses `npm run test:hardware` with `--screen-output` to display a pattern and save a screenshot; see *communication/validation*.

10.3.9 Reset, flashing and publication

```
await device.control({ action: 'reset', mode: 'soft', timeout: 15000 })
// await device.control({ action: 'reset', mode: 'hard' })
// await device.control({ action: 'bootloader' })
```

Core coordinates a new ready epoch and REPL barrier for soft reset. Inspect current state after timeout/cancellation rather than replaying business operations. Hard reset and downloader entry invalidate the link; an acknowledged request does not prove the rebooted device is online.

ROM flashing remains a separate lifecycle under `@esp-vision/device-core/firmware`. Hosts inject a ROM loader, approved images, chip/partition constraints and reset policy. Core ships no images and does not carry ROM download commands over EV-MUX. `control({ action: 'bootloader' })` only requests downloader entry; it does not flash. See *communication/interfaces* for physical handoff.

Package source, protocol tests and release automation are maintained together under `host/device-core/`. IDEs and custom applications consume pinned npm versions; agent device workflows use the Device CLI. SDK-only changes run JS tests, schema/package checks and publication dry runs without a firmware build matrix. Package releases are independent of firmware versions; publication jobs and `npm view` establish registry availability. Version/capability negotiation and hardware validation still determine actual compatibility; see *communication/validation* for acceptance evidence.

10.3.10 Firmware update application entry

Use `device.firmware.check()` to fetch the official release feed and per-version metadata automatically; `catalog` is an optional override for custom sources. Use `device.firmware.update({ release, openLoader, ... })` to install a selected compatible release. The packaged CLI supplies its Node ROM adapter internally; CLI users do not configure a loader script. See *Application API reference* for complete catalog, cancellation and reset contracts; direct FirmwareFlashSession remains available to loader adapters.

Node hosts import `openNodeRom` from `@esp-vision/device-core/node` and pass `openLoader: ({ signal }) => openNodeRom({ port, signal })`; no custom loader implementation is needed. Browser hosts provide `openLoader` for their Web Serial environment. Set `mode: 'install'` on check and update to provision a blank board; the default updates only the application.

10.3.11 Browser ROM flashing

SDK 0.6.0 provides shared runtime serial and ROM adapters through the `/browser` entry point. Use webpack, Vite or esbuild in a Web Serial-capable browser over HTTPS or localhost. No Python installation, host-maintained tasmota adapter or package-manager patch is needed.

```
import { createDevice } from '@esp-vision/device-core'
import { openBrowserRom } from '@esp-vision/device-core/browser'

// Request permission inside the click handler, before unrelated awaits.
const port = await navigator.serial.requestPort()
// Release any runtime serial connection before opening the ROM adapter.
const result = await createDevice().firmware.update({
  release: selectedRelease, board: selectedRelease.board,
  mode: 'update', // Explicit install requires qualified complete-image metadata.
  openLoader: ({ signal }) => openBrowserRom({ port, signal, logger: console }),
  onProgress: percent => updateProgress(percent),
})
if (result.needsReset) showManualReset()
```

`selectedRelease` comes from `firmware.check` for the physical board. Core downloads and validates before acquiring the ROM port, validates chip/revision/security before stub startup, checks existing layout (update) or measured capacity (install), and verifies every written byte. Never replace missing metadata with guessed offsets or silently switch update to install. Keep the page and port alive until the Promise settles: cancellation applies before writing and cannot interrupt a started write/readback. For a separate chip-detection UI, retain one adapter, await `initialize()`, then pass that adapter as `openLoader`; close it in `finally` even if download/metadata validation fails before Core acquires it. Native USB may require physical reset after verified flashing. This ROM adapter is separate from runtime EV-MUX serial transport and TCP OTA.

SDK 0.6.0 pins official `esptool-js@0.7.1`. It negotiates UART download speeds before compressed writes, handles direct USB-OTG ROM synchronization and validates bounded readback packets, their trailing MD5 and the complete write-range SHA-256. Runtime UART negotiation remains a separate EV-MUX lifecycle. Unsupported stubs and malformed SLIP packets remain explicit errors. The package root and Node entry points do not load this browser backend.

10.4 Device management CLI

The standalone `@esp-vision/device-cli` npm package installs `esp-vision-device`. The CLI uses the public `@esp-vision/device-core` API and supports discovery, device management, Python execution, device files, capture, batch plans, and bounded event watches. Requires Node.js 22 or newer.

For plugin installation and AI-assisted discovery, flashing, development and verification prompts, see *Build vision applications with AI*.

10.4.1 Installation and host requirements

```
npm install -g @esp-vision/device-cli
esp-vision-device --help
esp-vision-device inspect
```

Serial access uses bundled npm native bindings, and flashing uses the shared esptool-js backend. No host Python, pyserial or Python esptool installation is required. Node.js 22+, OS device drivers and port permissions remain required. LAN discovery uses bundled multicast-DNS. The deprecated `--python` option is accepted but ignored. Backend-unavailable, access-denied and busy errors have separate guidance; generic serial errors do not establish a device fault.

10.4.2 Discovery and inspection

Without a target, `inspect` enumerates OS serial ports and LAN advertisements without opening serial ports or acquiring device sessions. USB descriptors are candidate hints, not firmware identity. USB serial candidates are

shown by default; `--all-ports` includes other OS-enumerated serial ports. `--interface` accepts comma-separated `all`, `serial`, `usb`, `usj`, `uart`, and `tcp` filters. An explicit `--port` bypasses discovery filtering. Linux `/dev/serial/by-id/...` aliases are accepted.

```
esp-vision-device inspect --json
esp-vision-device inspect --interface serial --all-ports
esp-vision-device inspect --interface usj,usb --probe
esp-vision-device inspect --select 1 --fields identity,capabilities,runtime
esp-vision-device inspect --port /dev/ttyACM0 --role usj
esp-vision-device inspect --tcp esp-vision-board.local:3232 --fields identity,
↪network,storage
```

`--probe` queries discovered candidates and `--select` connects to the selected result from that invocation. Both use passive EV-MUX acquisition; they do not send Ctrl-C to the running application. `--probe-timeout` defaults to 5000 ms per candidate and accepts up to 60000 ms. Inventory collection defaults to 3000 ms; LAN collection is capped at 10000 ms. Targeted inspection defaults to identity, capabilities, and runtime; `--fields` also accepts `network` and `storage`. A targeted inspect, configure, firmware check, capture, watch, and native `files write` use passive acquisition. Python execution, REPL-backed `files list/read/remove/mkdir`, control, firmware update, and all batch plans use the normal REPL connection probe and can send Ctrl-C. For an explicit serial target, `--role` accepts `usj`, `usb`, `uart`, or `unknown` and requires `--port`; the default is `unknown`.

Runtime fields `scriptRunning` and `run` reflect work tracked by the current SDK session. A fresh CLI process cannot use those fields to determine whether a detached program is still running on the device. `vmReady` means the VM is ready; it does not mean the VM is idle.

10.4.3 Configuration, control, and firmware

For first-time Wi-Fi setup, connection checks and switching to wireless access, see *Host integration, Wi-Fi configuration and recovery*.

```
esp-vision-device configure --port COM3 --ssid MyWiFi
esp-vision-device configure --port COM3 --ssid MyWiFi --password-stdin
esp-vision-device configure --port COM3 --network-enabled false
esp-vision-device configure --port COM3 --forget-wifi
esp-vision-device control --tcp 192.168.1.100 --action stop
esp-vision-device control --port COM3 --action reset --mode soft
esp-vision-device control --port COM3 --action bootloader
esp-vision-device firmware check --port COM3 --version 2026.10.15
esp-vision-device firmware check --board ESP32_P4X_EYE --json
esp-vision-device firmware update --port COM3 --version 2026.10.15
```

Configuration prompts for a hidden Wi-Fi password; `--password-stdin` reads it from standard input. There is no command-line password option. Configuration output does not include credentials. Reset requires `--mode soft` or `--mode hard`; an acknowledgement does not mean the device has reconnected.

Firmware checks use the official feed by default; `--catalog` accepts a local catalog file or HTTPS URL. An offline check needs `--board` and optionally accepts `--current-version`. Update requires a local `--port` and explicit `--version`; the built-in serial flasher uses `esptool-js@0.7.1`. TCP is not OTA. The default `--mode update` writes the application and preserves the existing layout and user data. `--mode install` requires `--board`. When the CLI bundles Core 0.7.0 or later, it clears internal `ffat` files and writes the complete image, which may also replace NVS settings; SD cards are not erased. Older CLI bundles retain their older installation behavior: the independently released CLI does not automatically acquire a newer Core. Inspect the bundled version before relying on filesystem clearing. Use a ROM-capable USJ/UART port for blank-board installation. `--image` supplies local firmware bytes and does not bypass release integrity checks.

10.4.4 Python execution

Choose exactly one Python input: inline `--source CODE`, local `--file LOCAL.py`, or device `--path DEVICE.py`. `--completion wait` is the default and waits for completion. A positive `--timeout MS`

cancels a waiting execution when it expires. `--max-output-chars` bounds captured output (16,384 by default, up to 1 MiB).

```
esp-vision-device python --source 'print(42)' --port COM3
esp-vision-device python --file ./main.py --port COM3 --timeout 30000
esp-vision-device python --path /main.py --completion start --port COM3
```

`--completion start` accepts only `--path` and acknowledges after the device VM enters its runner. This does not confirm application initialization or completion. The CLI releases the connection and exits; it does not keep watching or resume the script. EV-MUX pins Python stdout to the session that started the program. Use foreground `--completion wait` for live Python output; a later watch cannot reattach to a detached run's stdout. Use captures or device-written progress artifacts to verify a detached application.

Foreground Python execution and Python steps in a batch stream `python.output`, `python.log`, and `python.state` NDJSON events on stderr. Output is bounded by `--max-output-chars` with one truncation marker. The stdout result contains metadata, not a duplicate of streamed Python text.

10.4.5 Device files and capture

Device paths such as `/models` belong to the device, including on Windows. Invalid paths are rejected before connecting. Git Bash/MSYS can rewrite slash-prefixed arguments to Windows host paths; use PowerShell or prefix the individual command with `MSYS2_ARG_CONV_EXCL="*"` and supply verified Windows-form local paths. Quoting `/models` alone does not prevent shell conversion. Do not strip a drive prefix from a converted path or retry device operations blindly.

```
esp-vision-device files list --path / --recursive --port COM3
esp-vision-device files read /main.py --output ./main.py --port COM3
esp-vision-device files write /backup.bin --file ./backup.bin --port COM3
esp-vision-device files remove /old.bin --type file --port COM3
esp-vision-device files mkdir /data --recursive --port COM3
esp-vision-device capture camera --output ./frame.jpg --quality 80 --port COM3
esp-vision-device capture screen --output ./screen.jpg --port COM3
```

File reads and JPEG captures write binary data to the requested local `--output` and return path, byte count, and metadata. Files read/capture are capped at 16 MiB by default; `--max-bytes` may lower the cap. `files write` sends local bytes with the binary file RPC. Images and file contents are not embedded in JSON.

`capture camera` observes the application's latest camera frame; it is not the LCD image. Application drawing depends on firmware publication behavior: firmware advertising `debug.capture_frame.observed.v1` publishes an independent copy at acquisition, before later drawing, without triggering a new acquisition. `capture screen` captures the current LCD framebuffer, including drawing rendered there, and is the source for verifying LCD overlays and UI. It requires an initialized display. Camera/display dimensions may differ; neither command starts the application or initializes the sensor/display. The output's `source` identifies the requested source, without certifying the presence or absence of annotations.

10.4.6 Batch and watch

`batch PLAN.json` validates the entire `schemaVersion: 1` plan before connecting, executes operations sequentially over one connection, and stops at the first failure. A report lists completed operations and the failure. Supported operations are `python`, `files.list`, `files.read`, `files.write`, `files.remove`, `files.mkdir`, and `capture`. Python steps wait for completion. Local file paths are relative to the plan file; device paths are absolute. Batch uses the normal REPL connection probe because a plan may run Python or REPL-backed file operations.

```
{
  "schemaVersion": 1,
  "operations": [
    { "op": "python", "source": "print('hello')" },
```

(continues on next page)

(continued from previous page)

```
    { "op": "files.read", "path": "/boot.py", "output": "./boot.py" },
    { "op": "capture", "source": "screen", "output": "./screen.jpg" }
  ]
}
```

`watch --duration MS` requires `--port` or `--tcp` and a duration from 1 ms to 1 hour. It passively observes output, logs, state changes, and diagnostics available to the currently acquired session, then releases the connection. Watch does not launch scripts, reconnect, or reattach to a prior detached run's Python stdout. SIGINT/SIGTERM cancel the command and release ownership.

10.4.7 Output, timeouts, and cancellation

`--json` writes one `schemaVersion: 1` JSON envelope to stdout. Status, Python output, logs, firmware phases, cancellation, errors, and duration are NDJSON events on stderr. Binary content is always written to local files. `--timeout` accepts 1..600000 ms. It applies to inspect/configure/control and inventory collection; `--probe-timeout` sets the timeout for each discovered candidate probe. When explicitly supplied, `--timeout` also applies to foreground Python, file reads, captures, and those operations in a batch. It is not a connection timeout: device connection setup uses 30000 ms. Python `--completion start` rejects `--timeout`; watch uses `--duration`; firmware operations do not use this option but support signal cancellation. During firmware update, once writing begins, Core completes readback verification and reset handling. File list/write/remove/mkdir support signal cancellation without a per-operation timeout. SIGINT exits with code 130, SIGTERM with 143, argument/plan errors with 2, other failures with 1, and success with 0. Busy or uncertain operations are not retried automatically. See `esp-vision-device --help --json` for the machine-readable command and capability reference.

10.5 Installation and compatibility

```
npm install '@esp-vision/device-core@~0.6.0'
# Optional Node LAN discovery adapter:
npm install multicast-dns
```

These examples target SDK `0.6.0`; declare a minor-locked range, record the resolved patch in a lockfile, and verify registry availability before installation. Node.js 22 is the host baseline. Browsers require a bundler/import map and a host-provided Web Serial or binary WebSocket transport. Only EV-MUX v4 is supported; a v3 device must update firmware. Package, firmware and IDE versions are independent. Source 0.6.0 is a minor release after 0.5.1; source changes and dry runs do not establish npm availability or update the Core bytes embedded in existing CLI/Plugin releases. Camera observation requires firmware feature `debug.capture_frame.observed.v1` and rejects older unsafe implementations locally.

10.6 Architecture and starting point

The same SVG is used by the project README and system architecture chapter. Host applications select devices and provide UI/workflow policy; Device Core owns the shared API, protocol and session state; EV-MUX defines runtime communication with the firmware. Byte adapters supply serial/socket I/O without parsing another protocol stack. Representative APIs in the figure do not replace the full API reference. Discovery and ROM flashing have separate lifecycles; the ROM loader does not carry image writes through EV-MUX.

The root exports only `createDevice`. It returns a device object; there is no public `Device` constructor. Use *Application API reference* for every application method, option, result and event. *Advanced export API reference* inventories all other package exports and their public class methods for integration maintainers. Agent workflows use the Device CLI for device operations; Knowledge MCP is a separate documentation interface.

```
import { createDevice } from '@esp-vision/device-core'
```

(continues on next page)

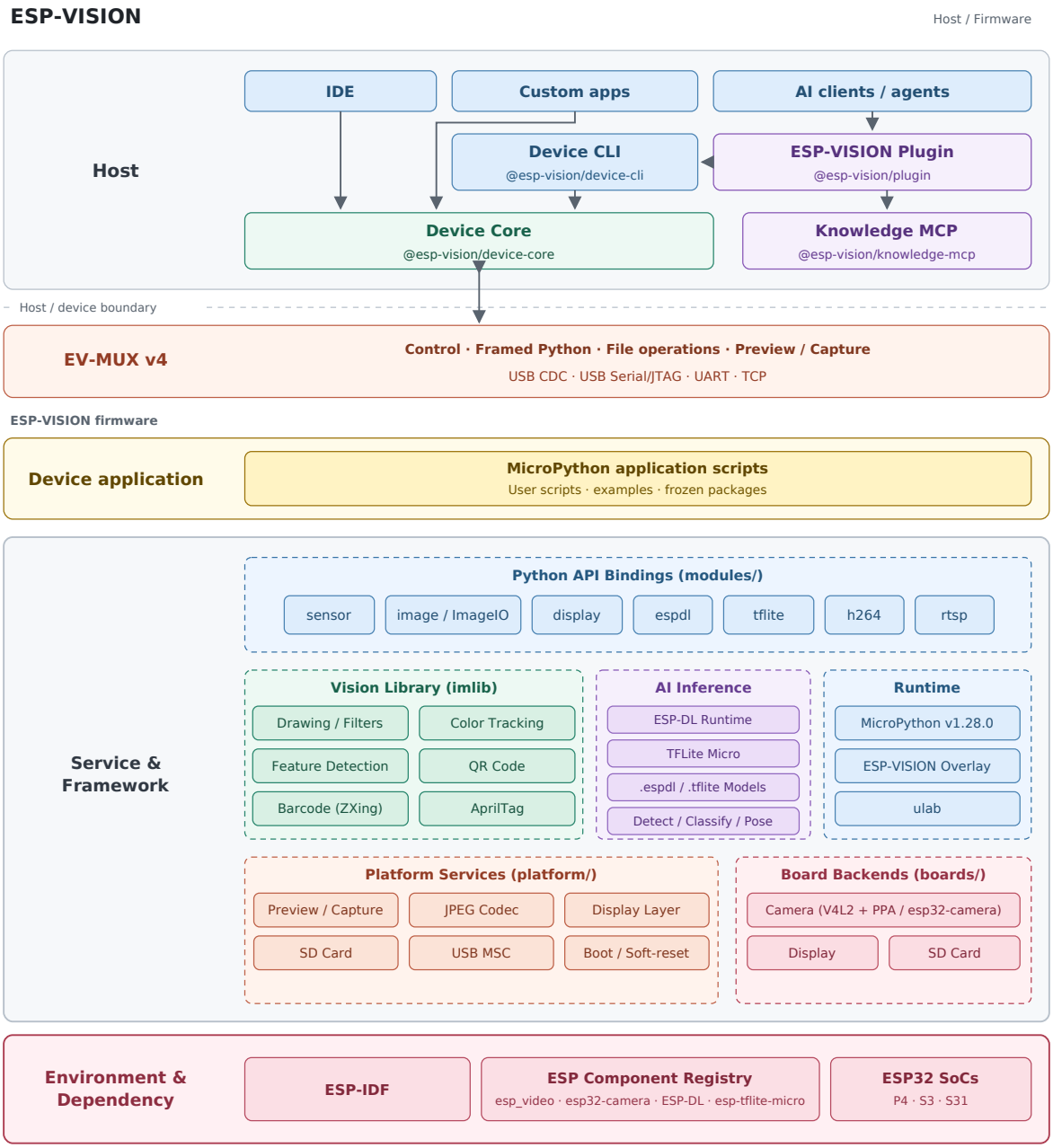


Fig. 2: IDE / Device CLI / custom host apps → Device Core → EV-MUX → firmware; Knowledge MCP → documentation over HTTP

(continued from previous page)

```
// transport is supplied by the host; its complete contract is in api.
const device = createDevice({ transport })
const off = device.on('python.output', text => console.log(text))
try {
  await device.connect({ role: 'usb' })
  console.log(await device.python.execute({ source: 'print(1 + 2)' }))
} finally {
  try { await device.disconnect() } finally { off() }
}
```

For complete runnable Node TCP and browser Web Serial adapters, see *Integrating the Device Core npm Package*. For packet layouts, routing and ownership, see *index*; hardware verification uses the shared entry point described in *validation*.

10.7 Licensing

The npm package and ESP-VISION contributions follow Apache-2.0, matching the firmware. The package includes LICENSE and NOTICE; NOTICE preserves historical MIT copyrights and permission text. Third-party dependencies retain their own licenses. See *Licensing* for the project inventory.

SDK 0.5.1 includes Node transports, serial/network inventory, ROM flashing, immutable release snapshots and single-BIN application updates. The `@esp-vision/device-core/node` entry exports these host adapters and bundler asset URLs. Installed checks: `npm --prefix node_modules/@esp-vision/device-core test` and `npm --prefix node_modules/@esp-vision/device-core run test:hardware -- --help`. A missing or empty suite fails. See *Integrating the Device Core npm Package* for application `discover()` versus provider `list()`.

Chapter 11

Project Relationships

This chapter explains how ESP-VISION relates to MicroPython, OpenMV, and its major third-party components. For licensing and redistribution obligations, see *Licensing*.

11.1 ESP-VISION, MicroPython, and OpenMV

The relationship can be summarized as follows:

- **MicroPython provides the language runtime and firmware base.** ESP-VISION starts from a pinned MicroPython ESP32 port and adds `overlay/micropython/`, MicroPython user C modules, and ESP-IDF components. It does not implement a separate Python interpreter.
- **OpenMV is an upstream source for part of the vision implementation and API design.** ESP-VISION reuses a subset of OpenMV `imlib` algorithms and some `image` binding and helper code, preserving their original license and copyright notices.
- **ESP-VISION is an independent integration for Espressif chips.** Camera, display, storage, USB, codec, ESP-DL inference, TFLite Micro inference, and board support are integrated using ESP-VISION and ESP-IDF components. ESP-VISION is not a fork of the OpenMV firmware and does not include the complete OpenMV hardware abstraction, IDE, or feature set.

The `sensor` and `image` modules therefore follow the OpenMV API style to make script migration easier, but **matching API names do not imply complete behavioral or feature compatibility**. Support also depends on the chip, board, memory, peripherals, and build options. Use this guide's *API Reference* as the authoritative support reference.

11.2 Dependency Layers

The main code layers in an ESP-VISION firmware image are:

```
User scripts
   |
   +-- MicroPython runtime and ESP32 port
   |
   +-- ESP-VISION Python modules
   |      +-- OpenMV-style sensor / image APIs
   |      +-- display / imageio / espdl / tflite / h264 / rtsp
   |
   +-- Algorithms and middleware
   |      +-- OpenMV imlib subset
   |      +-- ulab / ZXing-C++ / ESP-DL / TFLite Micro
```

(continues on next page)

(continued from previous page)

```
|
+-- ESP-IDF, managed components, and board backends
```

Dependencies come from three locations:

- Git submodules under `lib/`, such as MicroPython, `ulab`, and ZXing-C++;
- third-party-derived code maintained under `components/` and parts of `modules/`;
- ESP Component Registry packages resolved from `idf_component.yml`. The exact component set and versions can vary with the ESP-IDF version, selected chip, and board.

For the version, local path, and license of each dependency, see the *Licensing* inventory.

Chapter 12

Licensing

This chapter is the single reference for how licensing works across ESP-VISION. It is developer guidance, not legal advice; the license text and source-file headers for the exact version in use remain authoritative.

ESP-VISION' s own code (`platform/`, most of `modules/`, `boards/`, and build files) is released under Apache License 2.0, recorded in the repository-level `LICENSE`. Vendored and third-party code keeps the license declared by its own source files, SPDX identifiers, or upstream license; the repository-level `LICENSE` does not override or replace those.

12.1 License Inventory

The following table lists the direct dependencies most relevant to redistribution. It is not a complete inventory of transitive dependencies.

Project or code	Local path	Role in ESP-VISION	License
esptool-js 0.7.1	`host/device-core/package.json` dependency; loaded lazily by `src/rom_loader.js`	Shared browser Web Serial / Node ROM backend; upstream `src/targets/stub_flasher/README.md` identifies esp-flasher-stub v1.2.2 (set 2)	Apache-2.0 (installed package `LICENSE`); stub binaries dual Apache-2.0/MIT. Installed as a dependency; per-instance adapters do not modify upstream source. Transitive pako retains its MIT/Zlib notices; atob-lite retains MIT.
@serialport/bindings-cpp 13.0.0; Koffi 2.16.3	`host/device-core/package.json` optional dependencies; CLI/plugin native bundles	Native serial access and Linux USJ modem controls; original npm prebuilds and loaders	MIT; original package licenses and Koffi Node API/cnoke notices retained in portable bundles
Device Core host SDK	`host/device-core`	EV-MUX v4 host communication, Python, resource files, discovery and images	Apache-2.0 (`host/device-core/LICENSE`); historical MIT notices in `NOTICE`
Device-side camera/LCD test fixture	`host/device-core/tests/fixtures/device-checks.py`	Hardware acceptance functions migrated from the firmware tests; included in the npm package for shared hardware checks	Apache-2.0 (original notice retained)
sync-pr-to-gitlab (6ea5db6)	`.github/actions/sync-pr-to-gitlab`	GitHub PR to GitLab MR synchronization, adapted to remove Jira-specific metadata; original Python copyright and license notice retained	Apache-2.0
MicroPython v1.28.0	`lib/micropython`	Python runtime and ESP32 port baseline	MIT
MicroPython execution and thread adaptations	`overlay/micropython/shared/runtime/pyexec.c`, `overlay/micropython/ports/esp32/mpthreadport.c`	Session origin capture and inheritance based on the pinned upstream files; original copyright and license headers retained	MIT
MicroPython composite USB descriptor adaptation	`overlay/micropython/shared/tinyusb/mp_usbd_descriptor.c` (v1.28.0 baseline)	Built-in CDC/MSC/UVC descriptors and speed variants; original upstream header preserved	MIT
OV2710 timer lifecycle adaptation	`tools/prepare_ov2710.py`, `platform/ov2710_lifecycle.inc` and verbatim `tools/tests/fixtures/ov2710/upstream.c`; esp_cam_sensor 2.4.0, commit `8fc931636ffa937c89c`	Checked build copy of `sensors/ov2710/ov2710.c` (SHA-256 `4a9af395151527b156b1d68` 33a8391a641a5 acknowledge callback shutdown before SCCB release, preserving the original header without editing the managed component	Apache-2.0
TinyUSB UVC transfer adaptation	`overlay/micropython/`	Checked build copy of `src/class/video/`	MIT

12.2 How to Interpret the Licenses

The repository-level `LICENSE` covers original ESP-VISION code and does not override third-party licenses:

- third-party files remain under their original licenses and retain their original copyright and license headers;
- a directory can contain multiple licenses: most `imlib` code is MIT while `apriltag.c` is BSD-2-Clause;
- permissive licenses such as Apache-2.0, MIT, and BSD generally allow combined distribution, but their copyright, license text, and notice requirements still apply;
- some Espressif components use licenses with product-scope conditions. For example, the current `esp_new_jpeg` license text limits use to Espressif products, so it should not be described only as unrestricted generic MIT;
- trademarks, patents, model files, and datasets may have separate terms that cannot be inferred from the source-code license.

Before distributing source or firmware, check at least:

1. the repository-level `LICENSE`;
2. the `LICENSE` in every Git submodule;
3. `managed_components/*/LICENSE` from the selected build;
4. SPDX identifiers, license headers, and copyright notices in modified or newly added source files;
5. provenance and terms for non-code assets such as models, fonts, and test images.

12.3 OpenMV and GPL Code Paths

Not every optional OpenMV upstream algorithm uses the same license. ESP-VISION sets `OMV_NO_GPL=1` in `micropython.cmake` and `components/imlib/CMakeLists.txt` so GPL-conditioned OpenMV code paths are not compiled. This switch only describes what the default build excludes; it is not an automatic license classification for newly added files.

When changing `imlib` feature switches, synchronizing OpenMV upstream code, or adding algorithms, inspect each file's license. Do not infer a license only from the directory containing the file.

12.4 Adding Third-Party Code

When adding a third-party package or source file to ESP-VISION:

1. pin a reproducible release or commit;
2. verify the license per file and its compatibility with the intended distribution;
3. preserve upstream copyright, SPDX identifiers, license text, and any required NOTICE;
4. record the source, version, local path, purpose, and license in this inventory;
5. verify that build switches actually exclude code that is not intended for distribution instead of relying on documentation alone;
6. obtain legal review before merging when licensing is unclear or imposes commercial-distribution constraints.

Native binary-upload tests use the system OpenSSL 3 libcrypto (Apache-2.0) as their SHA-256 oracle through a PSA test shim. This is a host-test dependency only; OpenSSL and the shim are not linked into firmware or distributed in the npm package. Firmware uses ESP-IDF PSA Crypto.

Python Module Index

d

display, 69

e

espdl, 75

h

h264, 83

i

image, 53
imageio, 81

l

lcd_touch, 71

r

rtsp, 85

s

sensor, 49
storage, 87

t

tflite, 80

u

uvc, 86

Index

A

a_bins() (*image.histogram method*), 60
a_lq() (*image.statistics method*), 62
a_max() (*image.statistics method*), 62
a_mean() (*image.statistics method*), 62
a_median() (*image.statistics method*), 62
a_min() (*image.statistics method*), 62
a_mode() (*image.statistics method*), 62
a_stdev() (*image.statistics method*), 62
a_uq() (*image.statistics method*), 62
a_value() (*image.percentile method*), 61
a_value() (*image.threshold method*), 61
APPLY_COLOR_PALETTE_FIRST (*in module image*), 54
apriltag (*class in image*), 60
AREA (*in module image*), 53
area() (*image.blob method*), 58

B

b_bins() (*image.histogram method*), 61
b_lq() (*image.statistics method*), 62
b_max() (*image.statistics method*), 62
b_mean() (*image.statistics method*), 62
b_median() (*image.statistics method*), 62
b_min() (*image.statistics method*), 62
b_mode() (*image.statistics method*), 62
b_stdev() (*image.statistics method*), 62
b_uq() (*image.statistics method*), 62
b_value() (*image.percentile method*), 61
b_value() (*image.threshold method*), 61
backlight() (*display.ESP32Display method*), 70
barcode (*class in image*), 60
BAYER (*in module image*), 53
BICUBIC (*in module image*), 53
bilateral() (*image.Image method*), 66
BILINEAR (*in module image*), 53
BINARY (*in module image*), 53
binary() (*image.Image method*), 65
binary_to_grayscale() (*in module image*), 55
binary_to_lab() (*in module image*), 55
binary_to_rgb() (*in module image*), 55
binary_to_yuv() (*in module image*), 55
bind() (*in module uvc*), 86
bins() (*image.histogram method*), 60
BLACK_BACKGROUND (*in module image*), 54
blob (*class in image*), 57
blur() (*image.Image method*), 66
buffer_size() (*imageio.ImageIO method*), 82
bytearray() (*image.Image method*), 63

C

CENTER (*in module image*), 54
circle (*class in image*), 57
circle() (*image.circle method*), 57
classify() (*espdl.ImageNetCls method*), 78
clear() (*display.ESP32Display method*), 70
clear() (*image.Image method*), 64
close() (*h264.H264Encoder method*), 84
close() (*image.Image method*), 65
close() (*imageio.ImageIO method*), 82
CODABAR (*in module image*), 55
code() (*image.blob method*), 58
CODE128 (*in module image*), 55
CODE39 (*in module image*), 55
CODE93 (*in module image*), 55
compactness() (*image.blob method*), 58
compress() (*image.Image method*), 64
convexity() (*image.blob method*), 59
copy() (*image.Image method*), 63
CORNER_AGAST (*in module image*), 54
CORNER_FAST (*in module image*), 54
corners() (*image.barcode method*), 60
corners() (*image.blob method*), 57
corners() (*image.qrcode method*), 59
corners() (*image.rect method*), 57
count() (*image.blob method*), 58
count() (*imageio.ImageIO method*), 82
crop() (*image.Image method*), 63
cx() (*image.blob method*), 58
cxf() (*image.blob method*), 58
cy() (*image.blob method*), 58
cyf() (*image.blob method*), 58

D

data_type() (*image.qrcode method*), 59
DATABAR (*in module image*), 55
DATABAR_EXP (*in module image*), 55
deinit() (*display.ESP32Display method*), 69
deinit() (*espdl.ESPDet method*), 77
deinit() (*espdl.ImageNetCls method*), 78
deinit() (*espdl.Model method*), 76
deinit() (*espdl.NanoTrack method*), 76
deinit() (*espdl.PedestrianDetect method*), 77
deinit() (*espdl.YOLO11 method*), 78

deinit() (*espdl.YOLO11nPose method*), 78
deinit() (*lcd_touch.Touch method*), 71
deinit() (*tflite.Model method*), 80
density() (*image.blob method*), 58
detect() (*espdl.ESPDet method*), 77
detect() (*espdl.PedestrianDetect method*), 77
detect() (*espdl.YOLO11 method*), 78
detect() (*espdl.YOLO11nPose method*), 78
difference() (*image.Image method*), 65
dilate() (*image.Image method*), 65
display
    module, 69
draw_arrow() (*image.Image method*), 65
draw_circle() (*image.Image method*), 64
draw_cross() (*image.Image method*), 64
draw_edges() (*image.Image method*), 65
draw_ellipse() (*image.Image method*), 64
draw_image() (*image.Image method*), 65
draw_keypoints() (*image.Image method*), 65
draw_line() (*image.Image method*), 64
draw_rectangle() (*image.Image method*), 64
draw_string() (*image.Image method*), 64

E

EAN13 (*in module image*), 55
EAN2 (*in module image*), 54
EAN5 (*in module image*), 55
EAN8 (*in module image*), 55
ecc_level() (*image.qrcode method*), 59
eci() (*image.qrcode method*), 59
EDGE_CANNY (*in module image*), 54
EDGE_SIMPLE (*in module image*), 54
elongation() (*image.blob method*), 58
enclosed_ellipse() (*image.blob method*), 59
enclosing_circle() (*image.blob method*), 59
encode() (*h264.H264Encoder method*), 84
erode() (*image.Image method*), 65
ESP32Display (*class in display*), 69
ESPDet (*class in espdl*), 76
espdl
    module, 75
extent() (*image.blob method*), 58
EXTRACT_RGB_CHANNEL_FIRST (*in module image*), 54

F

FILE_STREAM (*in module imageio*), 81
find_apriltags() (*image.Image method*), 68
find_barcodes() (*image.Image method*), 68
find_blobs() (*image.Image method*), 67
find_circles() (*image.Image method*), 67
find_line_segments() (*image.Image method*), 67
find_lines() (*image.Image method*), 67
find_qrcodes() (*image.Image method*), 68
find_rects() (*image.Image method*), 68
flush() (*image.Image method*), 63
format() (*image.Image method*), 63
format() (*in module storage*), 87

G

gaussian() (*image.Image method*), 66
gaussian_blur() (*image.Image method*), 66
get_framesize() (*in module sensor*), 49
get_hist() (*image.Image method*), 67
get_histogram() (*image.Image method*), 66
get_hmirror() (*in module sensor*), 50
get_id() (*in module sensor*), 50
get_ipa() (*in module sensor*), 50
get_ipa_stats() (*in module sensor*), 50
get_isp() (*in module sensor*), 50
get_mode() (*in module uvc*), 86
get_percentile() (*image.histogram method*), 61
get_pixel() (*image.Image method*), 63
get_pixformat() (*in module sensor*), 49
get_regression() (*image.Image method*), 67
get_statistics() (*image.histogram method*), 61
get_statistics() (*image.Image method*), 67
get_stats() (*image.histogram method*), 61
get_stats() (*image.Image method*), 67
get_threshold() (*image.histogram method*), 61
get_vflip() (*in module sensor*), 50
GRAYSCALE (*in module image*), 53
GRAYSCALE (*in module sensor*), 49
grayscale_to_binary() (*in module image*), 55
grayscale_to_lab() (*in module image*), 55
grayscale_to_rgb() (*in module image*), 55
grayscale_to_yuv() (*in module image*), 56

H

h() (*image.barcode method*), 60
h() (*image.blob method*), 58
h() (*image.qrcode method*), 59
h() (*image.rect method*), 57
h264
    module, 83
H264Encoder (*class in h264*), 83
height() (*display.ESP32Display method*), 70
height() (*image.Image method*), 63
height() (*in module sensor*), 49
histeq() (*image.Image method*), 65
histogram (*class in image*), 60
histogram() (*image.Image method*), 67
HMIRROR (*in module image*), 53

I

I25 (*in module image*), 55
image
    module, 53
Image (*class in image*), 62
imageio
    module, 81
ImageIO (*class in imageio*), 81
ImageNetCls (*class in espdl*), 78
init() (*espdl.NanoTrack method*), 76
inputs() (*espdl.Model method*), 76

invert() (*image.Image method*), 65
IPAAFStats (*class in sensor*), 51
IPAAWBStats (*class in sensor*), 51
IPAStats (*class in sensor*), 51
is_alphanumeric() (*image.qrcode method*), 60
is_binary() (*image.qrcode method*), 60
is_closed() (*imageio.ImageIO method*), 82
is_kanji() (*image.qrcode method*), 60
is_numeric() (*image.qrcode method*), 60
is_streaming() (*in module uvc*), 86
ISBN10 (*in module image*), 55
ISBN13 (*in module image*), 55

J

JPEG (*in module image*), 53
JPEG_SUBSAMPLING_420 (*in module image*), 54
JPEG_SUBSAMPLING_422 (*in module image*), 54
JPEG_SUBSAMPLING_444 (*in module image*), 54
JPEG_SUBSAMPLING_AUTO (*in module image*), 54

K

keyframe() (*h264.H264Encoder method*), 84

L

l_bins() (*image.histogram method*), 60
l_lq() (*image.statistics method*), 62
l_max() (*image.statistics method*), 62
l_mean() (*image.statistics method*), 62
l_median() (*image.statistics method*), 62
l_min() (*image.statistics method*), 62
l_mode() (*image.statistics method*), 62
l_stdev() (*image.statistics method*), 62
l_uq() (*image.statistics method*), 62
l_value() (*image.percentile method*), 61
l_value() (*image.threshold method*), 61
lab_to_binary() (*in module image*), 56
lab_to_grayscale() (*in module image*), 56
lab_to_rgb() (*in module image*), 56
lab_to_yuv() (*in module image*), 56
laplacian() (*image.Image method*), 66
lcd_touch
    module, 71
length() (*image.line method*), 56
line (*class in image*), 56
line() (*image.line method*), 56
load_model() (*in module espdl*), 75
lq() (*image.statistics method*), 62

M

magnitude() (*image.circle method*), 57
magnitude() (*image.line method*), 57
magnitude() (*image.rect method*), 57
major_axis_line() (*image.blob method*), 59
mask() (*image.qrcode method*), 59
max() (*image.statistics method*), 62
mean() (*image.Image method*), 66
mean() (*image.statistics method*), 61
median() (*image.Image method*), 66
median() (*image.statistics method*), 61
MEMORY_STREAM (*in module imageio*), 81
midpoint() (*image.Image method*), 66
min() (*image.statistics method*), 62
min_corners() (*image.blob method*), 57
minor_axis_line() (*image.blob method*), 59
mode() (*image.Image method*), 66
mode() (*image.statistics method*), 61
Model (*class in espdl*), 76
Model (*class in tflite*), 80
module
    display, 69
    espdl, 75
    h264, 83
    image, 53
    imageio, 81
    lcd_touch, 71
    rtsp, 85
    sensor, 49
    storage, 87
    tflite, 80
    uvc, 86
morph() (*image.Image method*), 66

N

NanoTrack (*class in espdl*), 76

O

offset() (*imageio.ImageIO method*), 82
open() (*image.Image method*), 65
outputs() (*espdl.Model method*), 76

P

PALETTE_DEPTH (*in module image*), 53
PALETTE_EVT_DARK (*in module image*), 53
PALETTE_EVT_LIGHT (*in module image*), 53
PALETTE_IRONBOW (*in module image*), 53
PALETTE_RAINBOW (*in module image*), 53
payload() (*image.barcode method*), 60
payload() (*image.qrcode method*), 59
PDF417 (*in module image*), 55
PedestrianDetect (*class in espdl*), 77
percentile (*class in image*), 61
perimeter() (*image.blob method*), 58
pixels() (*image.blob method*), 58
PNG (*in module image*), 53
predict() (*espdl.Model method*), 76
predict() (*tflite.Model method*), 80

Q

QQVGA (*in module sensor*), 49
qrcode (*class in image*), 59
quality() (*image.barcode method*), 60
QVGA (*in module sensor*), 49

R

r() (*image.circle method*), 57

read() (*imageio.ImageIO method*), 82
read() (*lcd_touch.Touch method*), 71
rect (*class in image*), 57
rect() (*image.barcode method*), 60
rect() (*image.blob method*), 57
rect() (*image.qrcode method*), 59
rect() (*image.rect method*), 57
reset() (*in module sensor*), 49
reset() (*in module uvc*), 86
retry() (*in module storage*), 87
RGB565 (*in module image*), 53
RGB565 (*in module sensor*), 49
rgb_to_binary() (*in module image*), 56
rgb_to_grayscale() (*in module image*), 56
rgb_to_lab() (*in module image*), 56
rgb_to_yuv() (*in module image*), 56
rho() (*image.line method*), 57
ROTATE_180 (*in module image*), 54
ROTATE_270 (*in module image*), 54
ROTATE_90 (*in module image*), 54
rotation() (*image.barcode method*), 60
rotation() (*image.blob method*), 58
rotation_deg() (*image.blob method*), 58
rotation_rad() (*image.blob method*), 58
roundness() (*image.blob method*), 58
rtsp
    module, 85
RTSPServer (*class in rtsp*), 85

S

save() (*image.Image method*), 63
scale() (*image.Image method*), 63
SCALE_ASPECT_EXPAND (*in module image*), 54
SCALE_ASPECT_IGNORE (*in module image*), 54
SCALE_ASPECT_KEEP (*in module image*), 54
SEARCH_DS (*in module image*), 54
SEARCH_EX (*in module image*), 54
seek() (*imageio.ImageIO method*), 82
send() (*rtsp.RTSPServer method*), 85
sensor
    module, 49
SensorStatus (*class in sensor*), 51
set_frame() (*in module uvc*), 86
set_framesize() (*in module sensor*), 49
set_hmirror() (*in module sensor*), 50
set_ipa() (*in module sensor*), 50
set_isp() (*in module sensor*), 50
set_pixel() (*image.Image method*), 63
set_pixformat() (*in module sensor*), 49
set_thresholds() (*espdl.ESPDet method*), 77
set_thresholds() (*espdl.ImageNetCls method*), 79
set_thresholds() (*espdl.PedestrianDetect method*), 77
set_thresholds() (*espdl.YOLO11 method*), 78
set_thresholds() (*espdl.YOLO11nPose method*), 78
set_vflip() (*in module sensor*), 50
shutdown() (*in module sensor*), 49
size() (*image.Image method*), 63
size() (*imageio.ImageIO method*), 82
skip_frames() (*in module sensor*), 50
snapshot() (*in module sensor*), 50
solidity() (*image.blob method*), 59
statistics (*class in image*), 61
statistics() (*image.histogram method*), 61
statistics() (*image.Image method*), 67
status() (*in module sensor*), 51
status() (*in module storage*), 87
stdev() (*image.statistics method*), 61
stop() (*rtsp.RTSPServer method*), 86
storage
    module, 87
sync() (*imageio.ImageIO method*), 82

T

TAG16H5 (*in module image*), 55
TAG25H7 (*in module image*), 55
TAG25H9 (*in module image*), 55
TAG36H10 (*in module image*), 55
TAG36H11 (*in module image*), 55
tflite
    module, 80
theta() (*image.line method*), 57
threshold (*class in image*), 61
to_bitmap() (*image.Image method*), 64
to_grayscale() (*image.Image method*), 64
to_ironbow() (*image.Image method*), 64
to_jpeg() (*image.Image method*), 64
to_png() (*image.Image method*), 64
to_rainbow() (*image.Image method*), 64
to_rgb565() (*image.Image method*), 64
Touch (*class in lcd_touch*), 71
track() (*espdl.NanoTrack method*), 76
TRANSPOSE (*in module image*), 53
type() (*image.barcode method*), 60
type() (*imageio.ImageIO method*), 81

U

UPCA (*in module image*), 55
UPCE (*in module image*), 55
uq() (*image.statistics method*), 62
uvc
    module, 86

V

value() (*image.percentile method*), 61
value() (*image.threshold method*), 61
version() (*image.qrcode method*), 59
version() (*imageio.ImageIO method*), 82
VFLIP (*in module image*), 53
VGA (*in module sensor*), 49

W

w() (*image.barcode method*), 60
w() (*image.blob method*), 58
w() (*image.qrcode method*), 59

w() (*image.rect method*), 57
width() (*display.ESP32Display method*), 70
width() (*image.Image method*), 63
width() (*in module sensor*), 49
write() (*display.ESP32Display method*), 70
write() (*imageio.ImageIO method*), 82
write() (*in module uvc*), 86

X

x() (*image.barcode method*), 60
x() (*image.blob method*), 58
x() (*image.circle method*), 57
x() (*image.qrcode method*), 59
x() (*image.rect method*), 57
x1() (*image.line method*), 56
x2() (*image.line method*), 56
x_hist_bins() (*image.blob method*), 59

Y

y() (*image.barcode method*), 60
y() (*image.blob method*), 58
y() (*image.circle method*), 57
y() (*image.qrcode method*), 59
y() (*image.rect method*), 57
y1() (*image.line method*), 56
y2() (*image.line method*), 56
y_hist_bins() (*image.blob method*), 59
YOLO11 (*class in espdl*), 77
YOLO11nPose (*class in espdl*), 78
YUV422 (*in module image*), 53
yuv_to_binary() (*in module image*), 56
yuv_to_grayscale() (*in module image*), 56
yuv_to_lab() (*in module image*), 56
yuv_to_rgb() (*in module image*), 56